DevFluence™ School of Professional Software Development Chillisoft

JavaScript Fundamentals

Version: 1.0

Revision Date: 12 Oct 2018

© Chillisoft, 2018

Contents

© Chillisoft, 2018

© Chillisoft, 2018

© Chillisoft, 2018

Introduction

This course covers the fundamentals of the JavaScript language up to and including ES6. It also covers unit testing in JavaScript using Jasmine because unit tests are a fantastic way to learn a language.

This course assumes prior programming experience in other languages, and so assumes knowledge of programming concepts such as branching, looping, variables, scope and objects. It is intended as a cross-skilling course for those needing to add JavaScript to their existing programming toolset.

Key Texts

Although the content here has been created from scratch by referencing the ES6 definition, the style of the content in these notes follows the style of Douglas Crockford's *JavaScript: The Good Parts*. His book came out in 2008 and is a fantastic JavaScript primer; we would use it as the text of this course except it is unfortunately now out of date as the ES5.1 and ES6 specs have been published since then. These notes are intended to be up to date to the ES6 (ES2015 spec).

The ES5.1 spec can be found here: https://www.ecma-international.org/ecma-262/5.1/

The ES6 (ES2015) spec can be found here: https://www.ecma-international.org/ecma-262/6.0/

Another great resource referenced throughout is the Mozilla Developer Network's references pages on JavaScript: https://developer.mozilla.org/en-US/docs/Web/JavaScript

© Chillisoft, 2018

What is JavaScript?

"In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders" – Douglas Crockford, JavaScript: The Good Parts

JavaScript is one of the most popular programming languages in the world because a JavaScript engine is embedded in every browser. It is practically ubiquitous today, so it has become a tool that every developer should have in their toolbelt.

JavaScript originated in 1995 as a language embedded in Netscape Navigator 2.0, the market-leading browser at the time. In 1996 Netscape submitted JavaScript to ECMA International to try to create a standard specification for the language. The official name of the standard is ECMAScript, of which JavaScript is the main implementation. In 2009, ECMAScript 5 was released, and what is commonly called ECMAScript 6 (officially ECMAScript 2015) was released in 2015. There have since been updates in 2016 and 2017, but this course will not cover these releases.

Good Parts?

Due to the origination of JavaScript as an embedded language in a browser intended to add small pieces of dynamic functionality to web pages, and due to the speed it was put together and released, it has a lot of weird stuff in it. In this course, we will try to cover the parts of the language that we recommend using, not the full specification (if you'd like to read the specifications, see https://www.ecma-international.org/ecma-262/5.1/ for the ES5.1 and https://www.ecma-international.org/ecma-262/6.0/ for ES6/2015).

Playground

Having a place to play with a language is often the best way to really get to grips with it. These notes include many exercises that we recommend doing to explore the concepts covered in that section.

One recommended playground is Visual Studio Code with the Quokka extension. This will run the code in the editor and provide the output immediately so that you can easily inspect variables and see what is happening in your code.

© Chillisoft, 2018

Core Grammar

Note: these notes do not present the full grammar of JavaScript, only the grammar of the parts you should use – this is necessarily a subset of the full language definition. They are not exhaustive in nature but do try to cover all the parts of the language you would need.

Important things to start

Strict Mode

A syntactic unit in JavaScript can be processed using either unrestricted or strict mode. These notes assume all code is being processed using strict mode. This is usually ensured by including a string literal directive `'use strict'` or `"use strict"` at the top of a file, but it is also assumed within many newer JavaScript constructs (like classes). Coding without strict mode is not recommended. Many tools automatically add this literal to files.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Truthy and Falsy

In JavaScript, all expressions evaluate to truthy or falsy – that is, if you use an expression within an if statement's condition clause, the if statement condition clause will evaluate to true if the expression is *truthy* and false if the expression is *falsy*.

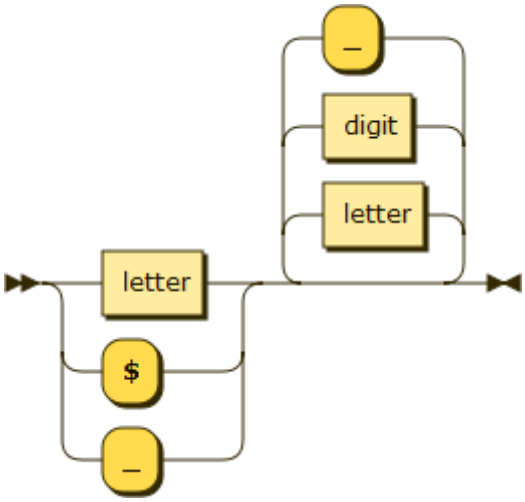
The *falsy* values are:

- `false`,
- `null`,
- `undefined`,
- the empty string (`''`),
- the number zero (`0`),
- the number `NaN`

Every other value is truthy. For example, the number `5` is truthy, the string `"Hello"` is truthy.

© Chillisoft, 2018

Identifiers

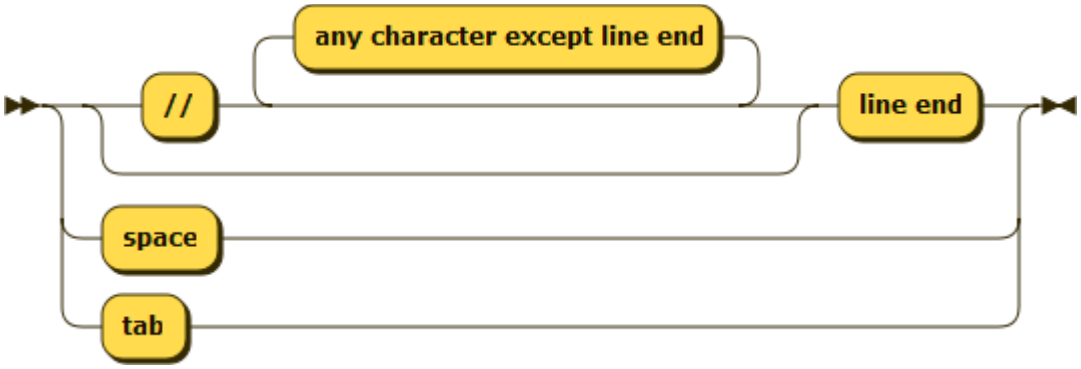


An identifier (or name) is a letter, $ or _ followed by one or more letters, digits or underscores. ES6 supports Unicode which extends the allowed characters, but we will not cover these extended characters here.

A name cannot be one of the reserved words – variables, parameters, object properties and labels are not permitted to use a reserved word as a name. The reserved words are:

```
await       enum        interface   this
break       export      let         throw
case        extends     new         true
catch       false       null        try
class       finally     package     typeof
const       for         private     var
continue    function    protected   void
debugger    if          public      while
default     import      return      with
delete      implements  super       yield
do          in          static
else        instanceof  switch
```

Whitespace (and comments)



Whitespace (the space, tab, or line end characters) separates tokens. Whitespace is not significant in JavaScript – any whitespace can be collapsed into a single space character, and in some cases can be removed altogether. It is only important when separating two tokens that cannot otherwise be separated. For example:

```
var x    =    'hello'      ;
```

The only required whitespace is the space between var and x so that these two tokens can be separated by the interpreter. All other whitespace can be removed. This is the same syntactically:

```
var x='hello';
```

Only one form of comment is recommended, starting with '//' and ending with the newline character. For example:

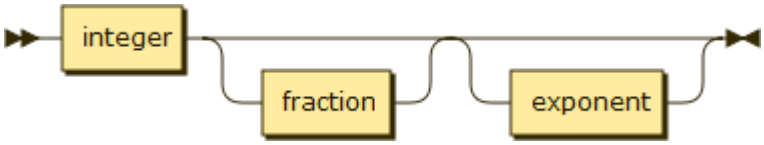
```
var x = 3.14159; // Approximately Pi
```

Another form of comment is supported – the block form starting with /* and ending with */, but these strings can appear within regular expression literals which can lead to unexpected results, so these are to be avoided.

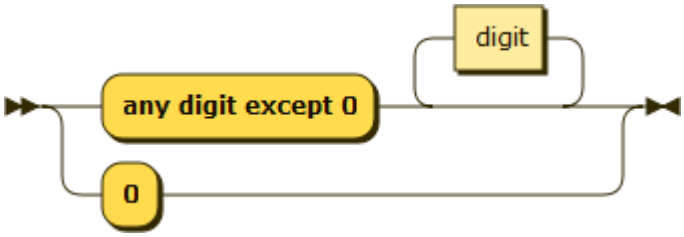
© Chillisoft, 2018

Numbers and Strings

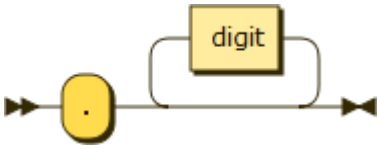
Numeric literal



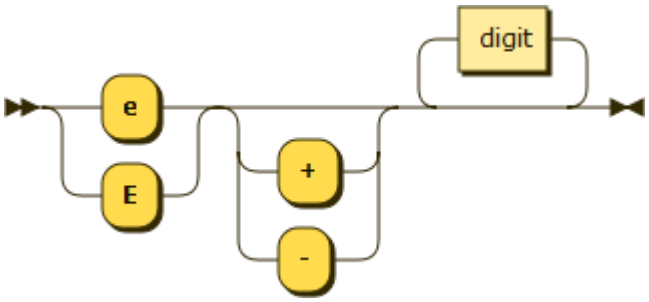
Integer



Fraction



Exponent



Examples

```
5
10.05
3.14159
5000
5e3     // 5000
0xFE0A
```

Description

There is only one number type in JavaScript – it is internally represented as a 64-bit floating point. This means that 1 and 1.0 are the same value. Numbers are objects that have methods.

A number literal consists of the integer part, an optional fraction part and an optional exponent part.

Negative numbers are indicated by the – prefix operator.

© Chillisoft, 2018

Octal, hex and binary integers are also supported; the grammar for this is left out here for simplicity's sake. Here are the basics:

- An octal integer has a leading `0` or a leading `0o` (e.g. `0o10`, which is 8 in decimal)
- A hexadecimal integer has a leading `0x` (e.g. `0x10`, which is 16 in decimal)
- A binary integer has a leading `0b` (e.g. `0b10`, which is 2 in decimal)

NaN

`NaN` is a special value – it is the result of an operation that cannot produce a number result. It is not equal to any value, including itself (`NaN == NaN` is a false statement). You can find out if a value is `NaN` by using the `isNaN(number)` function

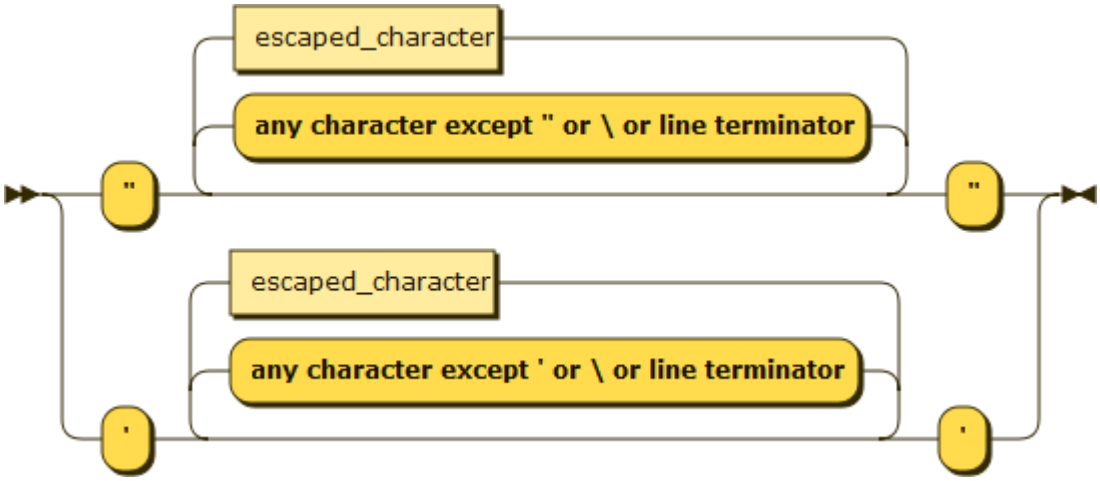
Infinity

`Infinity` is another special value – it represents all values greater than `1.7976931348623157e+308`

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates

String Literal



Examples

```
'hello'
"World!"
'hello' + "World!"
```

Description

String literals can be wrapped in single or double quotes. For example: `"use strict"` or `'use strict'` are both valid string literals.

Strings can contain zero or more characters, and all characters are 16 bits. The \ is the escape character, used for inserting characters that are not normally permitted, such as control characters or quotes (or backslashes).

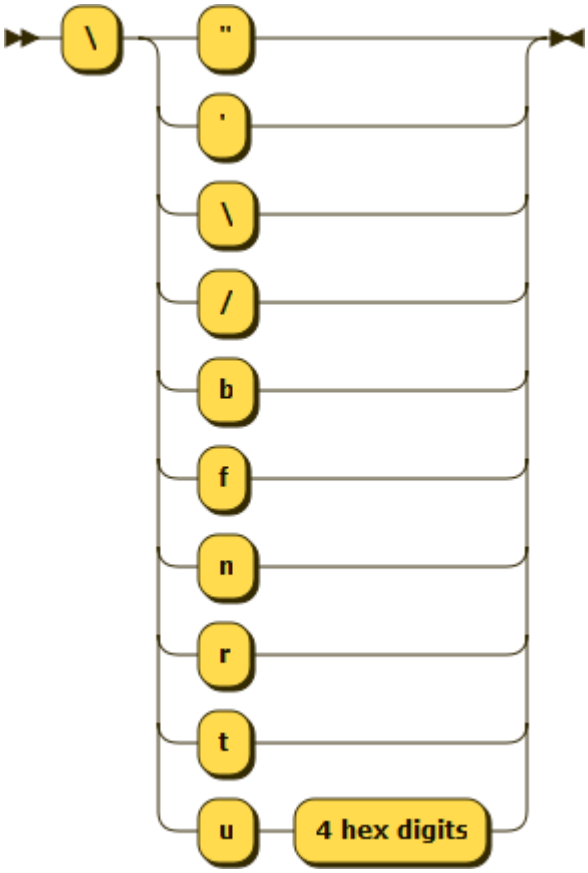
There is no character type in JavaScript.

Strings are immutable – once it is made, a string can never be changed.

Strings can be concatenated using the + operator.

```
'B' + 'o' + 'b' === "Bob" // Note that either single or double quotes can be used
```

Escaped character:



The above escape characters result in the following characters in the string literal:

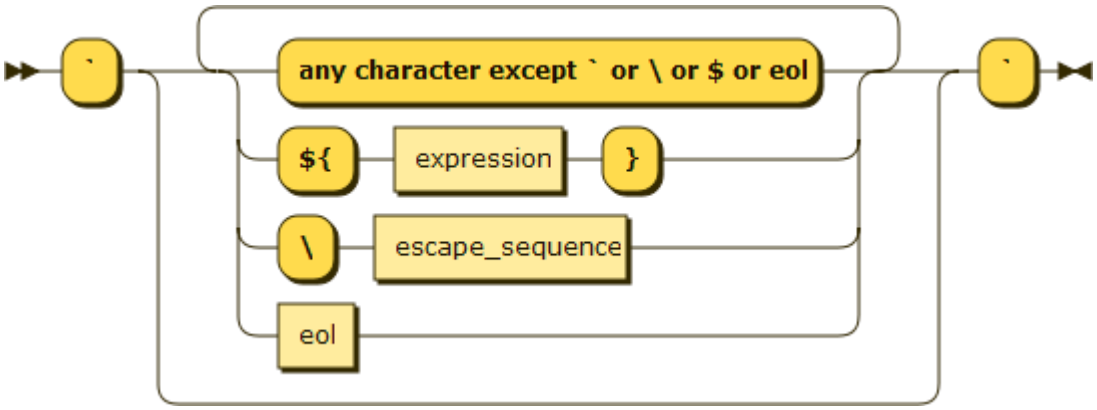
```
'\"' // the double quote character
'\'' // the single character
'\\' // the backslash character
'\/' // the forward slash character
'\b' // the backspace character
'\f' // the form feed character
'\n' // the new line character
'\r' // the carriage return character
'\t' // the tab character
'\u00A9' // the Unicode character © (with the hex value of 00A9)
```

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Text_formatting

© Chillisoft, 2018

Template Literals



Examples

```
let sbu = {
    name: 'Sbu',
    age: 45
}
let sbustr =
`Name: ${sbu.name}
Age: ${sbu.age}`
console.log(sbustr);
```

This outputs the following:

```
Name: Sbu
Age: 45
```

Description

JavaScript (from ES6) supports a newer form of string literal – the template literal. This is surrounded by backticks (the ` character). It supports interpolations using the ${ } characters as well as allowing end-of-line characters within the template literal.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Text_formatting

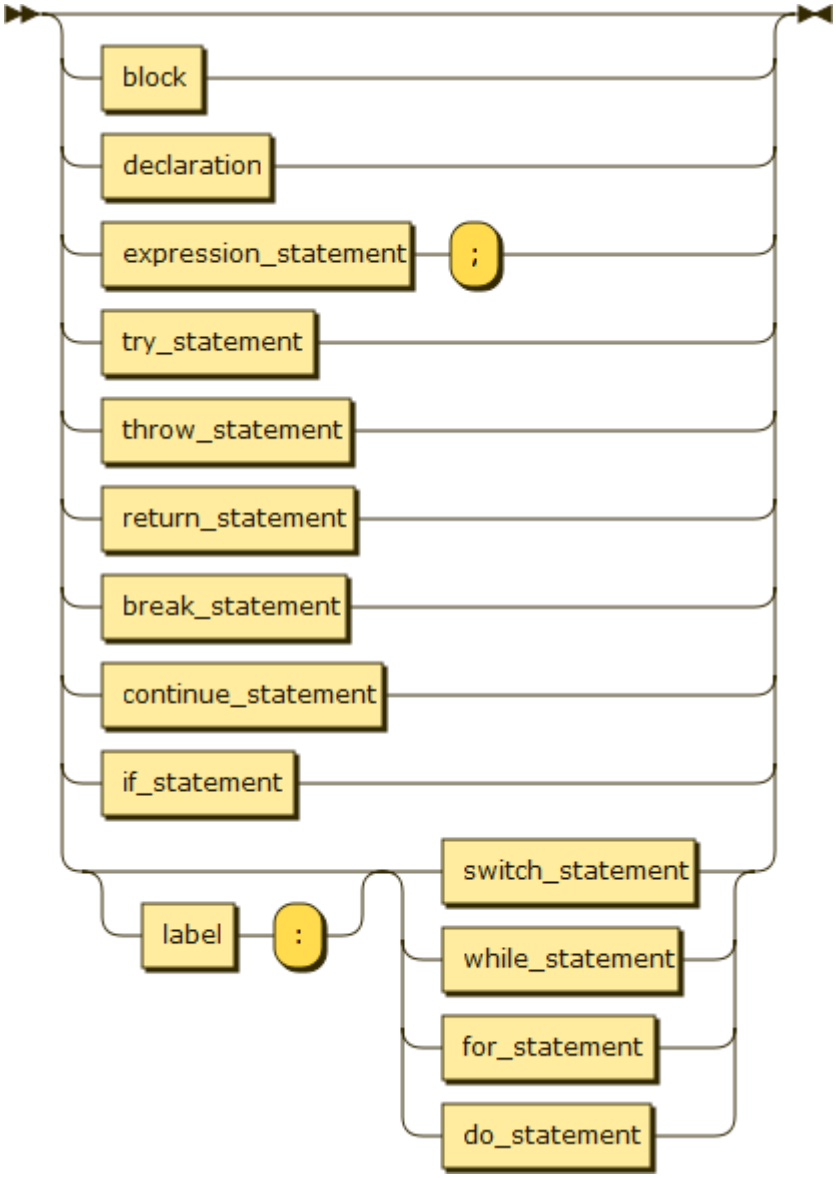
© Chillisoft, 2018

Statements

Now that we have covered the two main data types, numbers and strings, we can look at what statements we can use in JavaScript.

The various statements supported are shown in the diagram and detailed below.

Statement

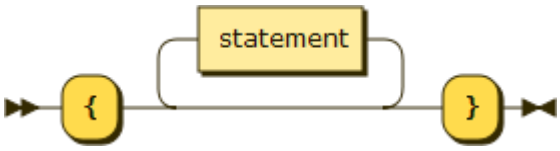


Any statement can be prefixed by a label, but this is only really useful for `break` and `continue` statements when indicating where to break or continue, so in the grammar described by the diagram a `label` only precedes one of the program flow statements that can contain `break` or `continue` statements.

A label follows the same grammar rules as an identifier.

© Chillisoft, 2018

Block



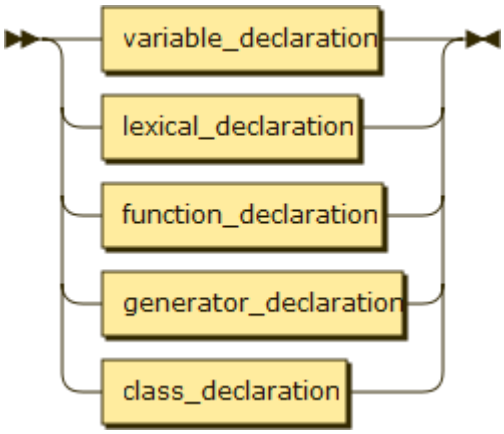
Examples

```
{
    console.log("Hello World!")
}
```

Description

A block defines a lexical scope within which declarations and other statements execute. When the block is exited the previous lexical scope is restored. See the Lexical declaration section on page 16 for more on lexical scopes.

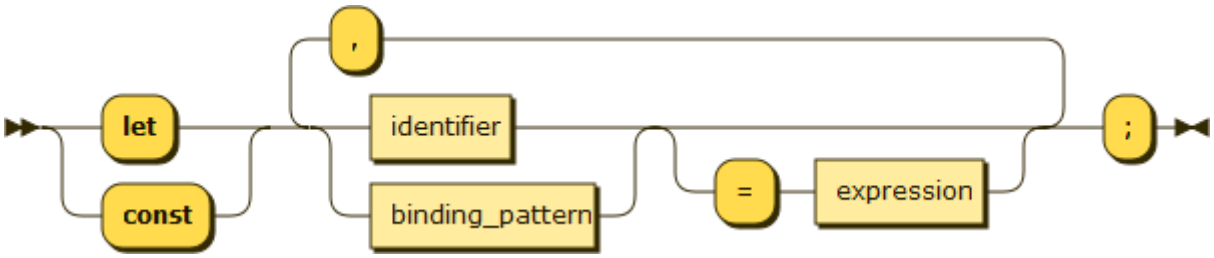
Declarations



A statement is a syntactic unit that expresses an action to be carried out. A program is essentially a sequence of one or more statements. Statements in JavaScript may contain declarations, expressions or literals.

© Chillisoft, 2018

Lexical declaration



Examples

```
let x = 20;
let junior = { name: 'Junior' };
let [a, b, c] = [1, 2, 3];
    // a === 1, b === 2, c === 3
let [d, e, ...f] = [1, 2, 3, 4, 5];
    // d === 1, e === 1, f === [3, 4, 5]
const [{g, h}, {i, j}, ...k] = [{g:1,h:2}, {i:3,j:4}, 5,6]
    // g === 1, h === 2, i === 3, j === 4, k === [5, 6]
let {l, m, n:[o,p,...q]} = {l:1,m:2,n:[3,4,5,6]}
    // l === 1, m === 2, n is not defined, o === 3, p === 4, q === [5, 6]
```

Description

There are four declaration keywords: `var`, `let`, `const` and `class`. This section covers `let` and `const`.

The `let` and `const` statements declare a variable that is scoped within the declaring block:

```
let x = 5;
{
    let x = 4;
}
x === 5;  // true - the x declared within the block is out of scope

{
    const y = 10;
}
console.log(y); // undefined
```

The example above shows that the variable declared within a { } block goes out of scope when the block is exited.

The `const` keyword is used in the same way as `let`, but the variable cannot be reassigned to another value – if you do this you will get an `"Assignment to a constant variable"` error.

New in ES6 is the concept of binding patterns which allow for destructuring assignments – see the next section on this.

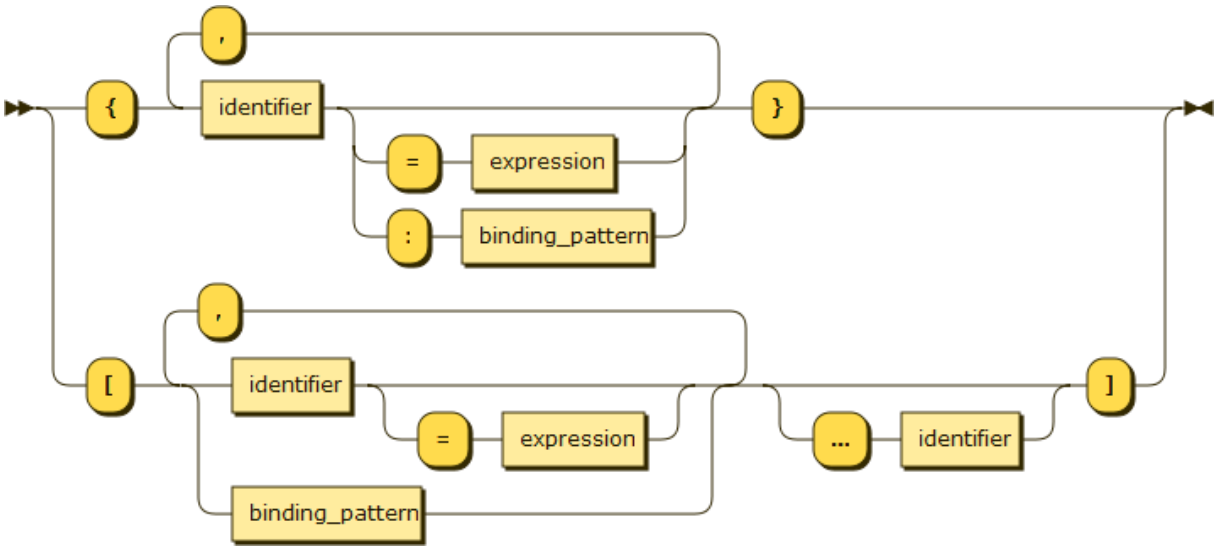
Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const

© Chillisoft, 2018

Binding pattern



Examples

```
let input = {
    firstName: 'Harry',
    dob: { y: 1979, m: 2, d: 15 },
    amounts: [10, 20, 30, 40, 50, 60, 70]
};
let {
    firstName,
    dob: dob,
    amounts: [amt1, amt2, amt3, ...remainingAmts],
    surname = 'Windsor'
} = input;
console.log(firstName, surname);              // Harry Windsor
console.log(dob);                             // { y: 1979, m: 2, d: 15 }
console.log(amt1, amt2, amt3, remainingAmts); // 10 20 30 [40, 50, 60, 70]
```

Description

Prior to ES6, only identifiers could be assigned to, but with ES6 on you can use a pattern to destructure an assignment. Some further examples of this are shown in the Lexical declaration section above. They can, however, be even more complex than these examples, as each binding pattern can contain assignments and also further binding patterns.

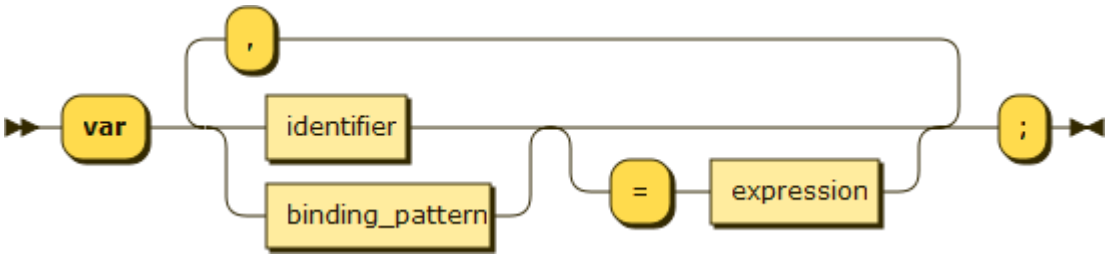
In the above example, the `let` statement is using a binding pattern to destructure the input object into a flatter structure where pieces of that object are assigned to different variables. Note that if the name of the variable in the binding pattern (e.g. `firstName` above) matches the name of the field in the object being assigned, then you don't need to indicate which field's value to use as it will use the matching field by default. You can specify if you want to – in the example the `dob` field is being assign explicitly to the `dob` variable.

For arrays you can also specify a variable within which to collect the remaining elements of the array (e.g `remainingAmts` in the above example) by using the ... operator (known as the rest operator).

Finally, note that you can also use normal assignment statements within this binding pattern – in the above example the variable `surname` is assigned the value `'Windsor'` within the binding pattern.

© Chillisoft, 2018

Variable declaration:



Examples

```
var x1 = 5;
var brian = { name: 'Brian' }
var [anne, bae] = ["Anne", "Bae"] // anne contains "Anne", bae contains "Bae"
var { pi, e } = { pi: 3.14159, e: 2.71828} // pi === 3.14159, e === 2.71828
```

Description

The `var` keyword declares a variable (or set of variables) scoped to the *running execution context's variable environment*. This means that `var` declarations are ***NOT lexically scoped***, that is the variable they declare does not only exist within the lexical block they are declared within. Instead they typically exist within the function they are declared, and, more than that, their declaration is hoisted to the top of that function. For this reason, **always prefer the use of `let`**, which is lexically scoped.

```
var y1 = 5;
{
    var y1 = 4;
}
y1 === 5; // false! The y1 declared within the block is the same y1
```

Compare this example with the `let` example above – if `let` were used then `y1` would be equal to 5, as expected, because the `y1` declared within the block would go out of scope at the end of the block. However, because var was used, the second `y1` declaration is hoisted to the top of the outer scope, and the line `var y1 = 4` is really just the same as saying `y1 = 4`.

```
{
    var z1 = 10;
}
z1 === 10; // true, z1 is hoisted up to the containing block
```

Once again, compare with the `let` example above – if `let` were used instead of `var`, then `z1` would be undefined in the outer scope. Because `var` was used, the declaration of `z1` was hoisted to the top of the outer scope and so it remains in scope even out of the block it was declared with.

For the above reasons, **always use `let`**! There is never a good reason to use `var` now that let provides proper lexical scoping.

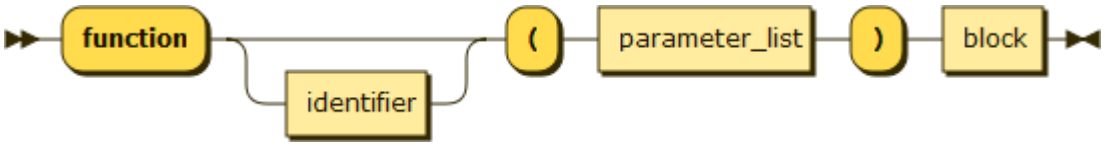
The `var` statement allows for destructuring in the same way that `let` and `const` do.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var

© Chillisoft, 2018

Function declaration



Examples

```
function square (x) {  return x*x; }
let sq4 = square(4);  // 16

let sq = function (x) { return x*x; }
let sq5 = sq(5);      // 25

let log = function (operand, base = e) {
    // calculate logarithm of operand to base provided
}
let log1000 = log(1000, 10); // 3

let calcAge = function ({y,m,d}) {
    // do calc using y, m and d
    return 39;
}
let dob = {y: 1979, m: 2, d: 15}
let age = calcAge(dob);  // 39
```

Description

A function declaration can include an identifier or not. If it does include an identifier (such as in the declaration of the square function in the examples) then it is a standalone declaration statement (as well as an expression that returns a function), and the function with the given identifier is scoped similarly to how `var` is scoped: the function declaration is hoisted to the top of the containing lexical scope (or the global scope if the function is declared in that scope).

If the function declaration does not include an identifier (such as the declarations of the functions assigned to `sq`, `log` and `calcAge`) in the examples, then the function declaration is an expression that can be used wherever expressions are allowed, and will need to be assigned to a variable to reuse later. *Always prefer this style of function declaration* because these functions are lexically scoped.

By doing an assignment in the parameter list you can also specify a default value for a parameter – see the `log` example above, where the base parameter is given a default value of `e` (where `e` is defined somewhere else).

Functions can be executed immediately:

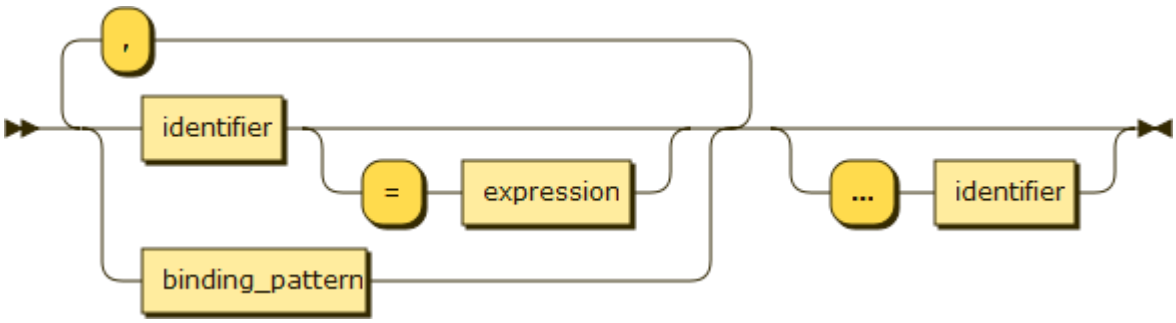
```
let result = function(input) {
    // do calculation of result
    return input + 5;
}(10); // result === 15
```

Note the `()` invocation at the end of the function definition – this runs the function immediately, so what is assigned to the `result` variable is the return value of the function, not the function itself.

© Chillisoft, 2018

Functions parameter lists can include binding patterns to allow for destructuring (see the `ageCalc` example above):

Parameter list

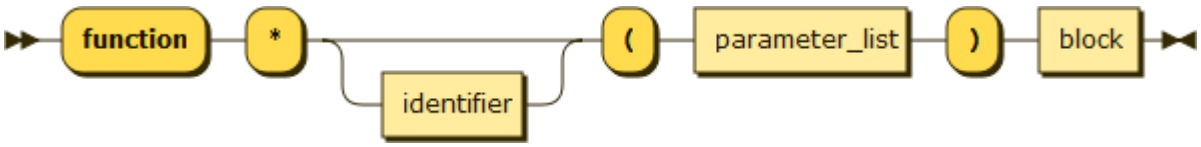


As in other forms of declaration, the parameter list of a function can include binding patterns to allow for destructuring of arguments passed in. It can also include assignments which provide default values, and an identifier that catches all remaining parameters in array form by using the … operator. Examples are provided in the Function declaration section above.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function

Generator declaration



Examples

```
// finite generator:
let greetingWordByWord = function * () {
    yield "Hello";
    yield "world!";
}
let greetingGen = greetingWordByWord();
let greeting = greetingGen.next().value + " " + greetingGen.next().value;
console.log(greeting);    // Hello world!

let furtherGreeting = greetingGen.next();
console.log(furtherGreeting.value);
console.log(furtherGreeting.done);

// infinite generator
let naturalNumbers = function * () {
    let num = 1;
    while (true) {
        yield num;
        num = num + 1
    }
}
```

```
let naturalNumberGen = naturalNumbers();
console.log(naturalNumberGen.next().value); // 1
console.log(naturalNumberGen.next().value); // 2, and so on

// generator that uses another generator
let generateTriangle = function * (size) {
    let x = 1;
    do {
        yield x;
        x += 1;
    } while (x <= size);
}
let triangleNumbers = function * () {
    let triangleSize = 1;
    do {
        yield * generateTriangle(triangleSize);
        triangleSize += 1;
    } while (true);
}();
console.log(triangleNumbers.next().value)
// continuing calls generate this sequence: 1 1 2 1 2 3 1 2 3 4 1 2 3 4 5
```

Description

Generator functions operate differently to regular functions in that they do not execute and return their result immediately; instead they return an iterator object. Then, when the `next()` method is called on that object, they execute until their next `yield` statement, and return the value generated by that `yield` statement. Later, when `next()` is again called on the iterator object, it continues executing from where it left off until the next `yield` is encountered. This continues until there are no more `yield` statements, at which point the result returned from the iterator has a value of `undefined` and its done property becomes `true`.

Generators can return infinite sequences – see the `naturalNumbers` example above.

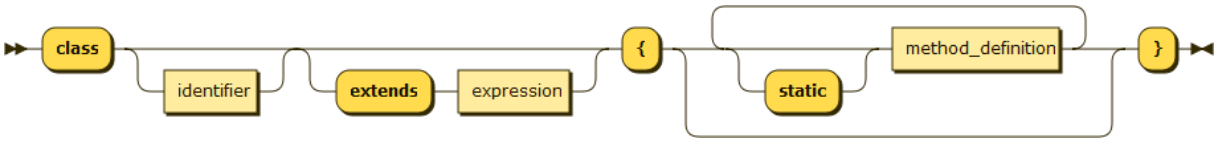
Generators can also use other generators buy using the `yield *` statement. See the `triangleNumbers` example above.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_generators

© Chillisoft, 2018

Class declaration



Examples

```
let Animal = class {
    constructor(name) {
        this.name = name;
    }
    speak() { console.log(this.name + " tries to speak"); }
}
let Cat = class extends Animal {
    constructor(name, colour) {
        super(name);
        this.colour = colour;
    }
    speak() {
        console.log(
            `${this.name} says miaow and licks her ${this.colour} fur.`);
    }
}
let a = new Animal("Wormy");
a.speak();  // Wormy tries to speak
let c = new Cat("McGee", "grey");
c.speak();  // McGee says miaow and licks her grey fur
```

Description

The class statement is really some syntactic sugar that makes creating object hierarchies in JavaScript easier to achieve. The result of class statement is, in fact, a function that creates an object.

In the example above, the Animal object is a function that creates an object that has a name property and a speak method (the word method is simply the name for a function located on an object). This function is called with the new keyword, which ensures that the 'this' reference within the function is correctly set up to refer to the object being created by the function instead of what it would normally be (more on 'this' can be found in the **Error! Reference source not found.** section on page **Error! Bookmark not defined.**.

The constructor method is a special one in the class declaration – the arguments passed into the new Animal call ("Wormy") in the example, are sent through to the constructor.

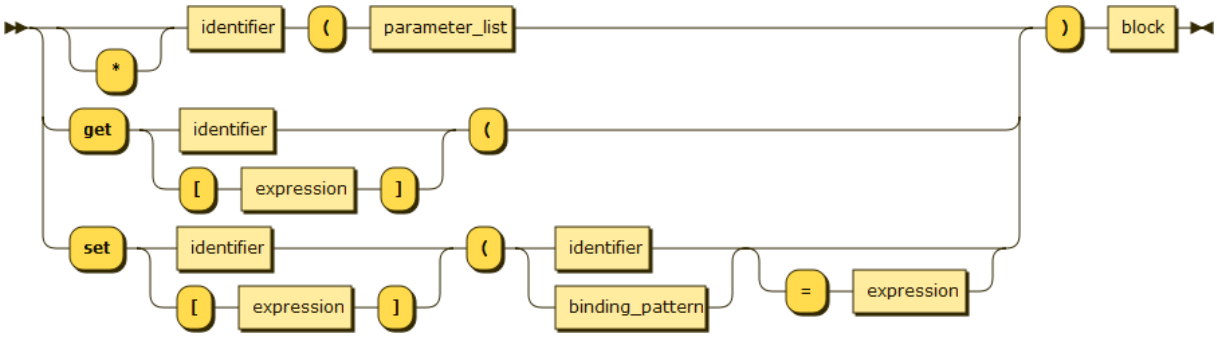
The extends keyword allows one class to inherit from another, utilizing JavaScript's prototypical inheritance system to do so (more on this can be found in the Objects section on page 42). When doing so, the super keyword allows methods to call the method of the prototype.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static

© Chillisoft, 2018

Method definition



Examples

```
class Rectangle {
    constructor(width, height) {
        this.setDimensions(width, height);
    }
    get width() { return this.w; }
    get height() { return this.h; }
    get ["perimeter" + "Length"]() {
        return (this.w + this.h) * 2;
    }
    set dimensions({width,height}) {
        this.setDimensions(width, height);
    }
    calculateArea() { return this.width * this.height; }
    setDimensions(width,height) {
        this.w = width;
        this.h = height;
    }
}
let r = new Rectangle(20, 10);
console.log(r.calculateArea()); // 200
r.setDimensions(20, 30);
console.log(r.calculateArea()); // 600
r.dimensions = {width:50, height:60};
console.log(r.width, r.height); // 50 60
console.log(r.perimeterLength);
```

Description

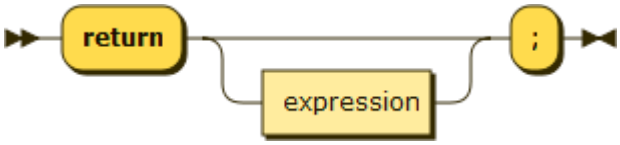
Classes can contain fields and methods – methods are the functions of the class. A method can be a generator function by prefixing its name with the * operator. Support for a property syntax is also provided by using the `get` and `set` keywords – these define methods that can be accessed as if they were fields (see the `width` and `height` getter and the `dimensions` setter in the example). Also allowed are calculated property names – surround the calculated name of a `get` or `set` method with `[ ]` to enable this.

Methods, like other functions, can include binding patterns that destructure the arguments passed in – the `dimensions` setter in the example is using this.

© Chillisoft, 2018

Control Statements

Return statement



Examples

```
let sq = function (n) {
    return n * n;
}
let sq10 = sq(10);
console.log(sq10);  // 100

let noReturn = function() { }
let result = noReturn();
console.log(result);  // undefined
```

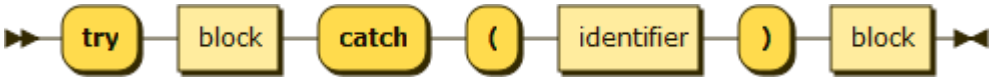
Description

The return statement returns control to the caller of the current function. It can also be used to specify the return value – if a return statement does not contain a return value expression then the return value will be undefined. A return statement can be placed anywhere within a set of statements. The statements after the return statement will not be executed as control is returned to the caller immediately.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/return

Try statement



Examples

See the **Error! Not a valid bookmark self-reference.** examples **Error! Not a valid bookmark self-reference.**.

Description

The try statement surrounds a block of code. If an exception is thrown by that block (through the use of a throw statement), then execution of that block will immediately end and control will be transferred to the block found in the catch clause. The object that is thrown will be stored in the identifier provided in the catch clause.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch

© Chillisoft, 2018

Throw statement

Examples

```
throw {
    name: "NotFoundException",
    message: "The expected object was not found."
}

let validateAge = function (age) {
    if (typeof(age) !== 'number') {
        throw {
            name: 'ValidationException',
            message: 'Age must be a number'
        }
    }
}
try {
    validateAge('5');
} catch (ex) {
    console.log(ex.name);     // ValidationException
    console.log(ex.message);  // Age must be a number
}
```

Description

Throw raises an exception. If the throw takes place within a try block then control is shifted to the catch clause of that try block. If it is not within a try block, then the function it is in is abandoned and control will go back to the catch clause of the outer function (and if that function has no try block then it will bubble up through the callstack until a catch clause is reached).

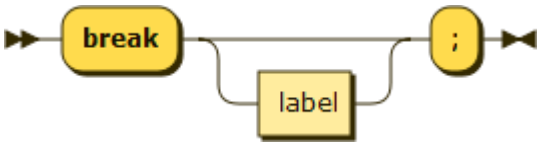
The expression after the throw can be anything, but is usually an object literal that contains a name and a message. The idea is to provide enough information for the caller to be able to do something with it.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw

© Chillisoft, 2018

Break statement



Examples

```
let str = "";
for (let i = 0; i < 4; i++) {
    for (let j = 0; j <= i; j++) {
        if (j === 1) {
            break;
        }
        str = str + j;
    }
    str = str + " ";
}
console.log(str);  // output is 0 0 0 0
```

```
let str = "";
outer_loop:
for (let i = 0; i < 4; i++) {
    for (let j = 0; j <= i; j++) {
        if (j === 1) {
            break outer_loop;
        }
        str = str + j;
    }
    str = str + " ";
}
console.log(str);  // output is 0 0
```

Description

The `break` statement causes an exit from a loop (`for`, `do` or `while`) or from a `switch` statement. It can optionally include a label that can specify the specific statement to break from (e.g. in the case of multiple loops.

In the first example above, the `break` statement drops execution out of the inner `for` loop into the outer `for` loop.

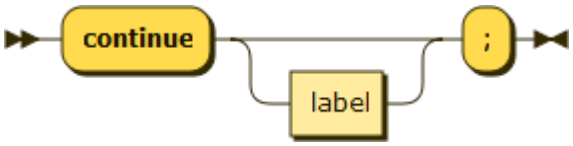
Using a label we can use the `break` statement to break out of the outer loop itself, as shown in the second example. This time when the `break` statement is encountered it breaks out of the *outer* `for` loop because the `outer_loop` label is indicated.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/break

© Chillisoft, 2018

Continue statement



Examples

```
let str = "";
for (let i = 0; i < 4; i++) {
    for (let j = 0; j <= i; j++) {
        if (j === 1) {
            continue;
        }
        str = str + j;
    }
    str = str + " ";
}
console.log(str);  // output is 0 0 02 023
```

```
let str = "";
outer_loop:
for (let i = 0; i < 4; i++) {
    for (let j = 0; j <= i; j++) {
        if (j === 1) {
            continue outer_loop;
        }
        str = str + j;
    }
    str = str + " ";
}
console.log(str);  // output is 0 000
```

Description

The `continue` statement has a similar control effect to the `break` statement, except it *continues* the loop instead of breaking out of the loop.

In the first example above, the case where `j === 1` is skipped but the loop continues to 2 and 3.

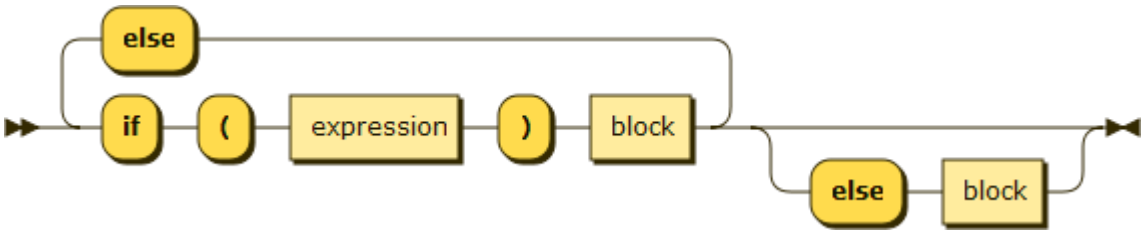
A `continue` statement can have an optional label, as shown in the second example.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/continue

© Chillisoft, 2018

If statement



Examples

```
let canDrive = function (age) {
    if (age < 18) {
        return false;
    } else if (age > 90) {
        return false
    } else {
        return true;
    }
}
console.log(canDrive(15));  // false
console.log(canDrive(100)); // false
console.log(canDrive(21));  // true
```

Description

An if statement can have a sequence of optional else if blocks. Note that a block starts with { and ends with }. While JavaScript supports single-line blocks without these parentheses, it's not recommended to use this because the interpreter may insert a semi-colon after the if statement but before the block:

```
if (x === 4)  // interpreter may insert a semi-colon here!
   x += 1;
```

If the interpreter does insert the semi-colon on the if line it changes the meaning significantly. To ensure this doesn't happen, always use a block:

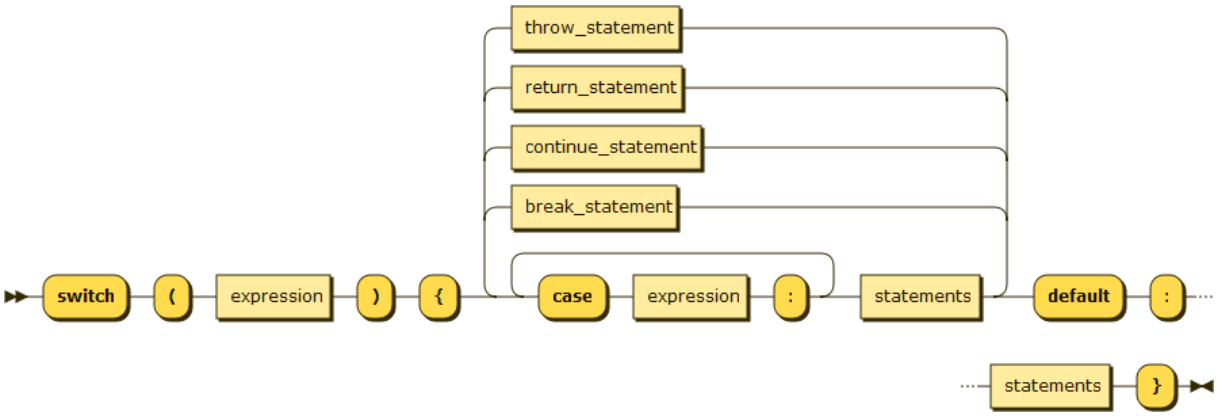
```
if (x === 4) {
   x += 1;
}
```

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else

© Chillisoft, 2018

Switch statement



Examples

```
let check = function (s) {
    switch (s) {
        case 4:
            s += 1;
            break;
        case 5:
            s = s * 5;
            break;
        case 6:
            s -= 1;
            // leaving out any control flow here is the same as continue;
        case 7:
            return 20;
        default:
            s = 2;
    }
    return s;
}
console.log(check(4));  // 5
console.log(check(5));  // 25
console.log(check(6));  // 20
console.log(check(7));  // 20
console.log(check(8));  // 2
```

Description

You can use a switch statement to perform a branch between multiple paths. The expression being switched on is compared with the expression in each case clause and if they match then the statements of that matching case clause are executed. If you do not include a break, return or throw after your statements then the following case will also be executed, even if it is not a match.

The default clause is used if no matches are found – this is optional.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch

© Chillisoft, 2018

While statement

Examples

```
let count = 1;
while (count < 10) {
    console.log(count);  // output is 1 3 5 7 9
    count += 2;
}

let obj = 3;
while (obj) {
    console.log("truthy"); // truthy truthy truthy
    obj -= 1;
}
```

Description

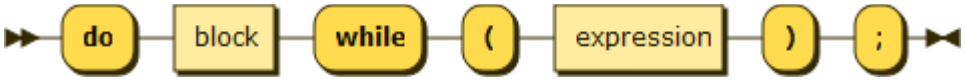
A `while` statement will execute the contents of its block as long as the specified expression is truthy. If the first time the expression is evaluated it is falsy then the block is not executed.

In the second example above, once the value of `obj` reaches `0` (zero) it becomes falsy (as `0` is a falsy value), which causes the `while` statement to end.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/while

Do statement



Examples

```
let count = 1;
do {
    console.log(count);  // output is 1 3 5 7 9
    count += 2;
} while (count < 10);

let obj = 3;
do {
    console.log("truthy"); // truthy truthy truthy
    obj -= 1;
} while (obj);
```

Description

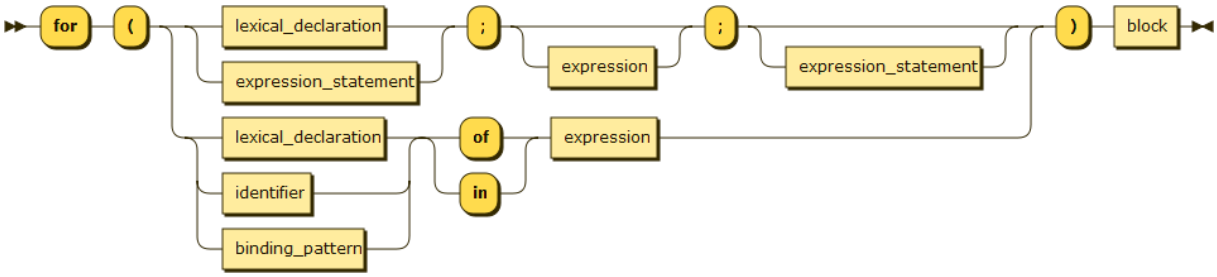
A `do` statement executes the block, and then checks the expression for truthiness. If it is truthy then it executes the block again, and continues until the expression evaluates to a falsy value.

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/do...while

© Chillisoft, 2018

For statement



Examples

```
for (let i = 0; i < 3; i += 1) {
    console.log(i);  // 0 1 2
}

let count = 0;
for (;;) {    // not a recommended use - rather use while (true) { ... }
    if (count === 3) break;
    console.log(count);  // 0 1 2
    count += 1;
}

let evensToN =  function * (n) {
    for (let even = 0; even < n; even += 2) {
        yield even;
    }
}
for (let even of evensToN(10)){
    console.log(even) // 0 2 4 6 8
}
```

Description

The `for` loop is a shorthand notation for a `while` loop; in it you define a starting condition, an ending condition and an increment expression statement. The loop will initialise with the starting condition and then while the ending condition is truthy it will run the contents of the block, ensuring that the increment expression statement is run and the ending condition is checked for truthiness before executing the block again.

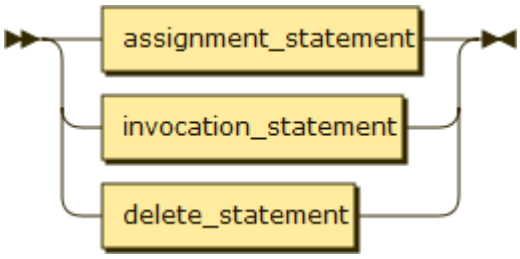
There is a `'for of '` form which iterates through all the items in an iterable object, including: string, array, Array-like objects, TypedArray, Map, Set, and user-defined iterables. This is the most convenient way to iterate through a list of items but care should be made not to confuse it with the `'for in '` form of this statement.

The `'for in '` form iterates through all the property names of an object. This is only usually useful for reflection-type operations and is one of those not-so-good parts of JavaScript, so it is not covered here.

Further references

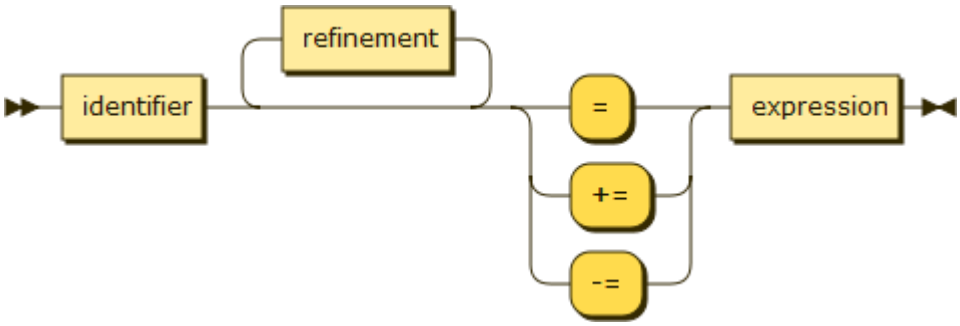
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for

Expression statement



The expression statement is one that assigns a value to a variable, invokes a method or calls the delete keyword. These result in a value (hence they are expressions) but they can also be statements on their own.

Assignment statement



Note: Refinement refers to the use of the . or [ ] operators to select fields within an object –

Examples

```
let bob = { name: 'Bob', age: 25 }
bob.age = 29;   // bob.age is now 29
bob.age += 1;   // bob.age is now 30
let age = bob.age -= 1;        // bob.age is now 29, so is age
let newAge = bob['age'] += 10; // bob.age is now 39, so is newAge
```

Description

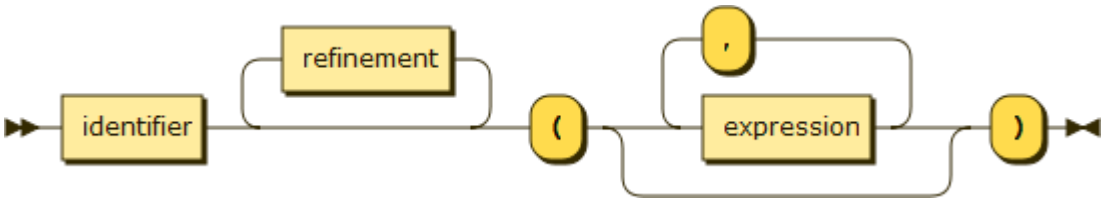
The single equals ('=') is the assignment operator; it assigns the value of the right hand side expression to the variable indicated by the identifier on the left. The left hand side can

Also supported are += and -= which are shorthand assignment operators and the preferred operators for incrementing or decrementing. Refinement refers to the . or [] operators that allows you select properties on the object to assign to (please see the Refinement section on page 37.)

There are other assignment operators: *=, /= and %=. These are seldom used but are available – we don't recommend their use over the long forms (x = x * y, x = x / y and x = x % y) because these forms are clearer to read – in general even use of the += and -= operators should be minimised other than for incrementing or decrementing by one.

© Chillisoft, 2018

Invocation statement



Examples

```
let subs = {
    name: 'Subs',
    age: 20,
    incrementAge: function() { this.age += 1; }
}
subs.incrementAge(); // subs.age is now 21
```

Description

An invocation is a statement that calls (invokes) a function on an object. Objects can be composed of other objects which have functions on them.

Delete statement

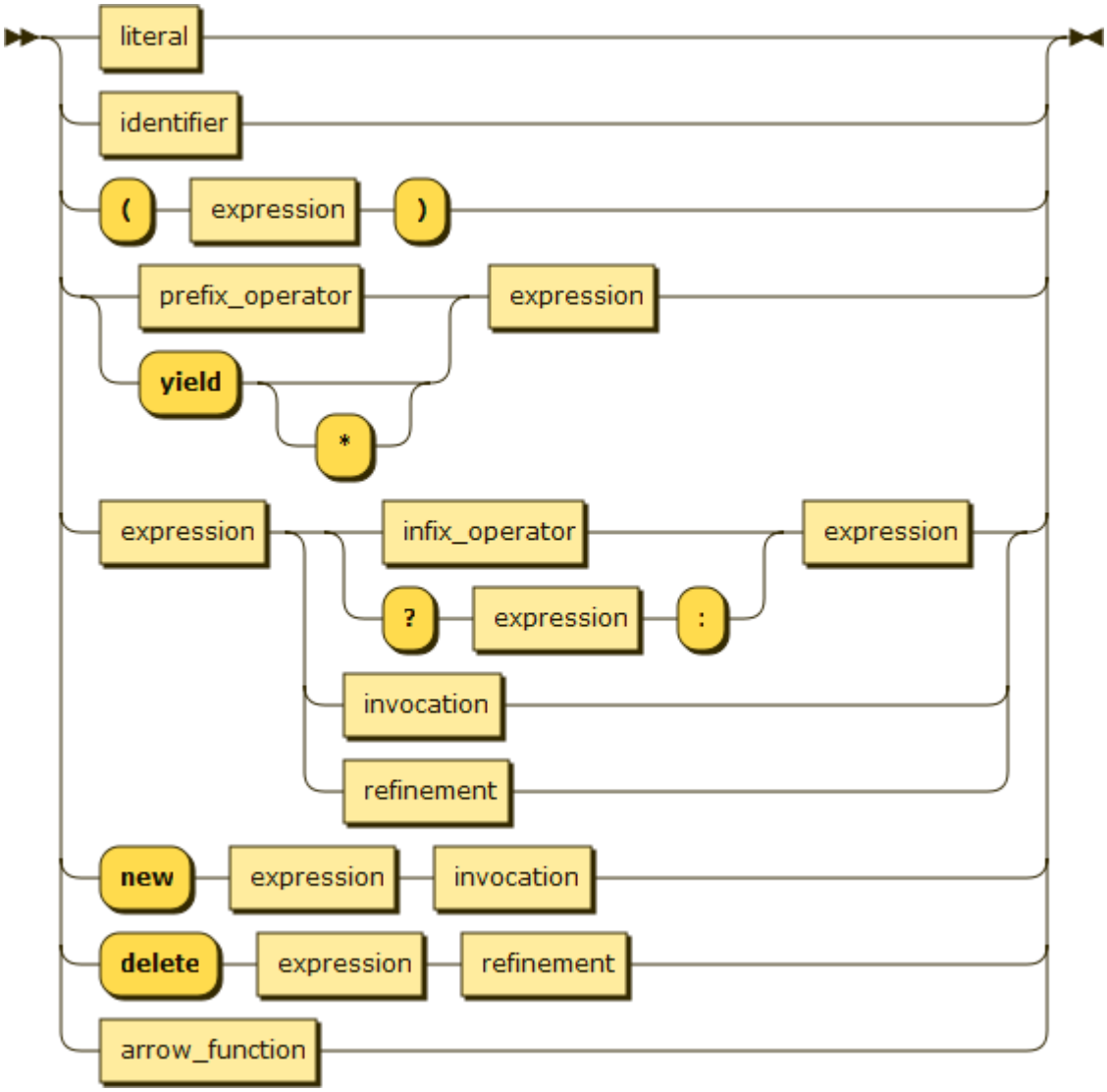
Examples

```
let amahle = {
    name: 'Amahle',
    age: 25
}
delete amahle.age;    // amahle.age is now 'undefined'
```

Description

The delete keyword can be used as a standalone statement – it deletes a property from an object.

Expressions



Expressions can be as simple as a literal or a variable name or built-in value. They always yield a value and do not usually make sense without being part of a statement that utilizes the resulting value.

Operators

JavaScript has prefix, infix and postfix operators. The only postfix operators are the ++ and – operators, which are not recommended due to the confusion they can create.

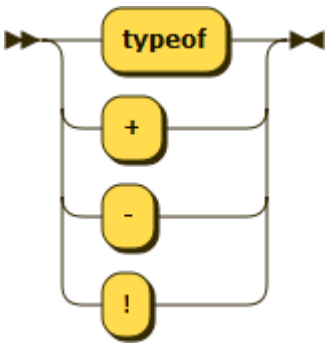
Operator Precedence

`. [] ()`	Refinement and invocation
`delete new typeof + - !`	Unary operators
`* / %`	Multiplication, division, modulo
`+ -`	Addition/concatenation, subtraction

© Chillisoft, 2018

>= <= > <	Inequality
=== !==	Equality
&&	Logical and
||	Logical or
?:	Ternary

Prefix Operators



Expressions can be combined with a prefix operator:

Symbol	Operator name	Details	Example
typeof	type of	Produces one of the following values: 'number', 'string', 'boolean', 'undefined', 'function', 'object'. (beware of caveats – typeof null produces 'object').	`typeof 3 // 'number'` `typeof '3' // 'string'`
+	To number	Converts the expression to a number type	`typeof '3' // 'string'` `typeof +'3' // 'number'`

-	Negate	Converts the expression to a number type and negates it	`typeof '3' // 'string'`
			`typeof -'3' // 'number'`
			`-'3' === -3; // true`

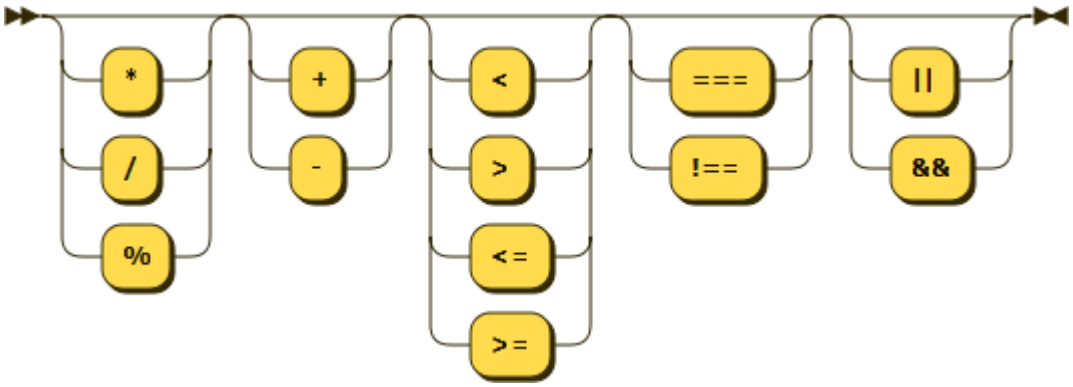
!	Logical not	If the operand is truthy, returns false, otherwise, returns true. The double not (!!) can be used to convert truthyness to true and falsyness to false.	`!'hello' // false`
			`!!'hello' // true`
			`!null // true`
			`!!null // false`

Also defined in the specification are the pre-increment and pre-decrement operators (++ and --). We do not recommend using these due to their ability to create confusion. For this same reason we have excluded the postfix operators (++ and -- after an expression). Generally, prefer the `'+= 1'` option as this is much more readable. This is, however, a question of style and your team's preference may vary.

© Chillisoft, 2018

Also included in the specification is a bitwise not operator (~). Bitwise operators are not recommended either – see the section within Infix Operators on this.

Infix Operators



Note: These operators are represented above in groups to indicate order of precedence. Only one operator can be used at a time.

Examples

```
let x = 5;
let y = 10;
let z = x + y;  // 15
let greeting = 'Hello' + ' ' + 'world!';  // 'Hello world!'
let happy = true;
let healthy = false;
let isOK = happy && healthy;  // false
let isPartiallyOK = happy || healthy;  // true
```

Description

Infix operators are placed between two expressions and perform the operation on these two expressions. The diagram above indicates the order of operator precedence – also see the Operator section on page 34.

Not shown in this diagram are the == and != form of the equal and not equal operators. This is for good reason – these should never be used. Instead, **always use the === and !== form** of these. The == and != form will do type conversion which can lead to weird things:

```
'3' == 3;   // true, which is an odd result
'3' === 3;  // false, which is expected
```

The Bitwise Operators

The bitwise shift operators (<<, >> and >>>, as well as the assignment ones >>=, <<= and >>>=) are also supported, but left out here. In most languages, using bitwise operators to do arithmetic operations or bit manipulation is really fast, but not in JavaScript. Remember that all numbers in JavaScript are stored as double-precision floating point, but that bitwise operations only work on integers – this means that JavaScript will convert the value to a 32-bit integer, perform the bitwise operation and then convert the value back. Unless you are literally modelling bitwise arithmetic, don't use these operators.

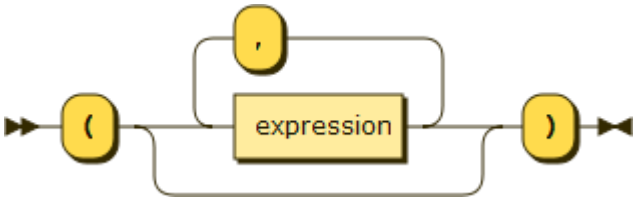
Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators

© Chillisoft, 2018

Other Expressions

Invocation



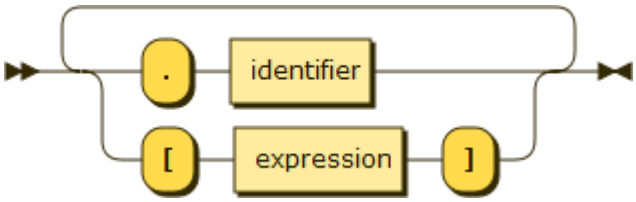
Examples

```
let f = function(x) {
    return 5 * x;
}
console.log(f(10));
```

Description

Invocation is the calling of a function or method on an object.

Refinement



Examples

```
let rectangle = {
    width: 20,
    height: 30
}
let width = rectangle.width;            // 20
let height = rectangle['hei' + 'ght']; // 30
let area = rectangle.area;              // undefined
```

Description

Refinement refers to the accessing of properties on an object using either the . or [ ] operators. The . operator requires an identifier, while the [ ] operator can be used with an expression which results in a string – the resulting string will be the property that is accessed. In either case, if the property does not exist then the result will be undefined.

New and delete

For a description of delete, please see the Delete statement section on page 33.

The new keyword can be placed ahead of an invocation as a modifier. It affects the way in which the function is called. For a full description of this, please see the **Error! Reference source not found.** section on page **Error! Bookmark not defined.**.

© Chillisoft, 2018

Ternary Operator

Examples

```
let greeting = function(hour) {
    return hour < 12 ? "Good morning" : "Good day";
}
console.log(greeting(8));  // Good morning
console.log(greeting(14)); // Good day
```

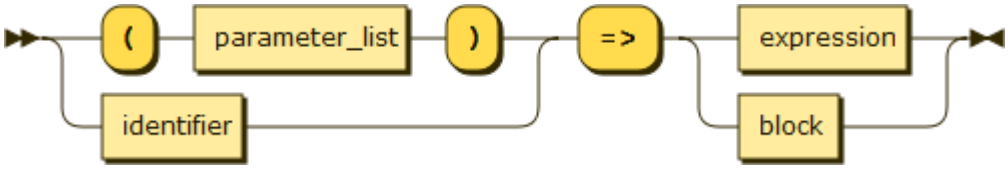
Description

The ternary operator (`?:`) is a shorthand for an `if-then-else` statement. The above example is the equivalent of the following:

```
let greeting = function (hour) {
    if (hour < 12) {
        return "Good morning";
    } else {
        return "Good day";
    }
}
```

It's pretty clear that the ternary option is much terser than the if-then-else format. If you are not used to the ternary syntax it can be tricky to read, but it is common enough across all C-derivative languages that it is a staple of simple if-then-else constructs.

Arrow Functions



Examples

```
let double = x => 2 * x;
console.log(double(5)); // 10

let area = (width, height) => width * height;
console.log(area(10, 20)); // 200

let collect = (...test) => test;
console.log(collect(1,2,3)); // [1, 2, 3]

let multiply = ({x, y}) => x * y;
console.log(multiply({x:2,y:3})); // 6

let numbers = [1, 2, 3, 4, 5]
let squares = numbers.map(n => n * n);
console.log(squares); // [1, 4, 9, 16, 25]
```

© Chillisoft, 2018

Description

The arrow function is a shorthand for declaring a function and are especially suited to be used as arguments for functions that take functions as parameters (for example, `Array.map()` shown above, which executes the given function using each array element and returns a new array with the results).

Arrow functions are not the same as function expressions in behaviour – they don't have a `this` defined, they don't provide access to their arguments with the `arguments` object. For these reasons they're not suited to using to define methods on objects, and they're best kept to short functions, especially when passing functions as arguments.

Yield expression

Examples

For examples of `yield`, please see the Generator declaration section on page 20. The `yield` and `yield*` statements can only be used within generator functions (i.e. functions declared with `function*`)

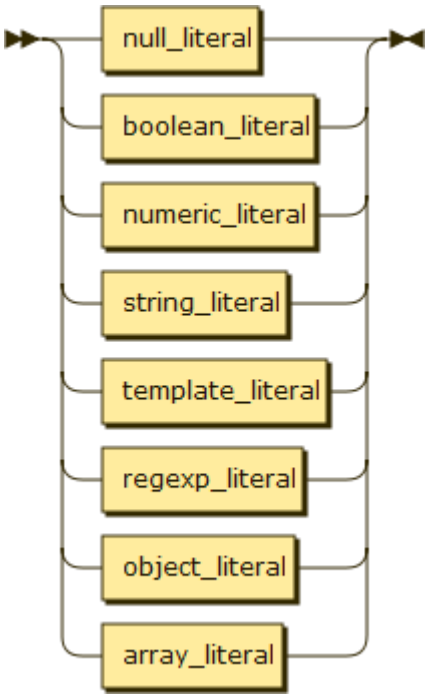
Description

The `yield` keyword is used to return a result from a generator function. Doing this pauses the function and returns control to the calling function until the next call to the `next()` method on the generator object's iterator.

The `yield*` keyword is used to delegate through to another generator function.

Please see the Generator declaration section on page 20 for more details on generators and `yield`.

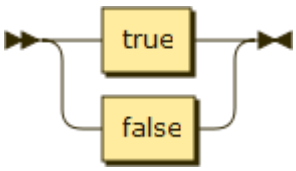
Literals



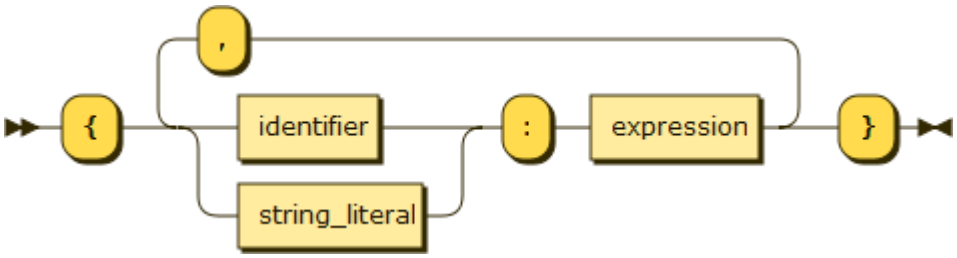
Literals can be used to specify numbers, strings, objects, arrays, functions or regular expressions. Literals evaluate to a value. We've covered the numeric, string and template literals above.

Null literal

Boolean literal



Object literal



Examples

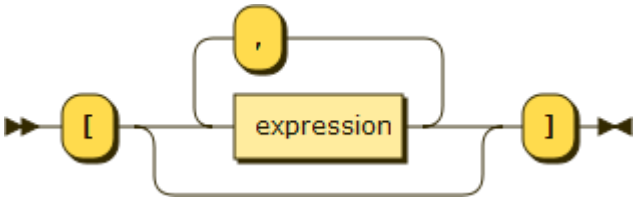
```
let options = {
    textColour : 'red',
    fontSize : 10,
    '452Option': 4523
}
```

Description

Object literals are a very convenient way to specify new objects. The names of properties can be specified as a name or using a string literal – names must be known at compile time. Here is an example:

Functions can also be included in object literals – anything that is an expression can be included on the right hand side of each object literal declaration clause.

Array literal



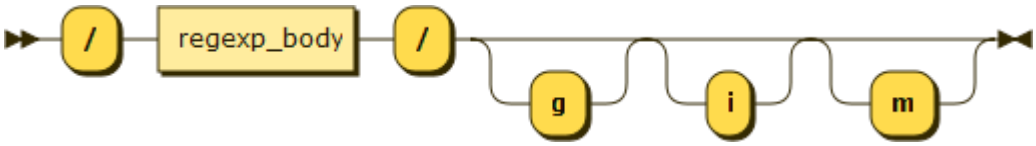
Examples

```
let values = [ 5, 3, 7.5, 2, 10 ]
```

Description

Array literals are a convenient way of specifying new arrays. For example:

Regular expression literal



Examples

```
let containsDigit = /[0123456789]/
console.log(containsDigit.test('1'));
console.log(containsDigit.test('a'));
console.log(containsDigit.test('1,000'));
console.log(containsDigit.test('1000'));
console.log(containsDigit.test('a1'));

let containsOnlyDigits = /^[0123456789]*$/
console.log(containsOnlyDigits.test('1'));
console.log(containsOnlyDigits.test('a'));
console.log(containsOnlyDigits.test('1,000'));
console.log(containsOnlyDigits.test('1000'));
console.log(containsOnlyDigits.test('a1'));
```

Description

"I once had a problem, so I used a regular expression to solve it, and now I have two." – unknown.

Regular expressions are a very powerful tool to use in the right circumstances, and JavaScript has direct support for them through the regular expression literal, which creates a RegExp object. The RegExp object has a bunch of methods that can test (check if a match occurs) or execute (return all matching substrings) them.

Regular expressions are not covered in more detail in these notes.

Reference

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Function literal

Please see Function declaration on page 19 – a function literal is merely the version of function declaration that includes the name of the function within the declaration itself.

© Chillisoft, 2018

Some examples of literals

Numeric literal	`3.14159`
	`100`
	`1.5e10`

String literal	`"Hello world!"`
	`"Welcome to Mark's world!"`
	`'Happy'`
	`'He said: "Hello world!"'`

Template literal	`` `Hi! My name is ${name}` ``

Object literal	`{`
	`  age: 25,`
	`  "surname": "Mkhize"`
	`}`

Array literal	`["one", "two", "three" ]` `[1, "two", 3]`

Function literal	`function square (x) {`
	`  return x * x;`
	`}`

Regular expression literal	`/[0123456789]/`

Further references

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_Types

© Chillisoft, 2018

Objects

In JavaScript, everything is an object, even the core types such as numbers and strings. Functions are objects too, complete with properties and methods.

Objects are always passed by reference, they are never copied:

```
let x = {}, y = x;
x.value = 5;
console.log(y.value); // 5
```

Objects have properties which are other objects. One really convenient way to create an object is to use the object literal notation.

```
let point = {
    x: 0,
    y: 0,
    distanceFromOrigin: function () {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
}
```

Access properties with the . or the [ ] operators:

```
point.x = 3;
point.y = 4;
console.log(point.distanceFromOrigin());  // 5
console.log(point['x']);   // 3
```

Properties can be functions, in which case they're called methods (e.g. distanceFromOrigin in the example above). To invoke functions, use the () operator. You can get the function object by using the property name. To invoke that function, you will need to use its apply() method, passing in the object to invoke it on:

```
let distanceFromOrigin = point.distanceFromOrigin;
console.log(distanceFromOrigin.apply(point));  // 5
```

Note: The apply() method will have no effect on an arrow function's this reference.

Prototypes

All objects have a prototype, accessible from the prototype property which is an object too. The default prototype object provides some properties and methods that are available to all objects.

Let's look at an example. We start by creating an object we want to use as a prototype:

```
let animalPrototype = {
    name: 'unnamed',
    speak: function() {
        console.log(`${this.name} made a noise.`);
    }
}
```

© Chillisoft, 2018

```
console.log(animalPrototype.hasOwnProperty('speak'));  // true
```

Here we define an original prototype object called animalPrototype, and it defines a name property and a speak method. We can use the hasOwnProperty() method, available on all objects, to check if a property belongs to the object or to its prototype. In this case the speak() method is defined on the object itself.

A prototype object can be specified when creating an object using Object.create().

```
let dog = Object.create(animalPrototype);
dog.speak();      // unnamed made a noise.
console.log(dog.hasOwnProperty('speak'));  // false
```

Once we create the dog object using the animalPrototype, we are able to call speak() on it. Note, however, that the dog object does not have a property on it called speak – this property is found on the prototype of the dog object.

When we call a method on an object, the JavaScript runtime will try to retrieve that property from that object, and if it is not found there, JavaScript will go to the prototype object to retrieve it; if it isn't there, it will go to the prototype's prototype, and so on until the chain ends or the property is found.

We can override the prototype's speak method by creating one on the dog object:

```
dog.name = 'Fido'
dog.speak();      // Fido made a noise
dog.speak = function() {
    console.log(`${this.name} barked!`);
}
dog.speak();      // Fido barked!
console.log(dog.hasOwnProperty('speak')); // true
```

Now when we call speak(), the runtime finds it on the dog object itself and calls it without even looking at the prototype object.

We can use this new dog object as the prototype for another:

```
let poodle = Object.create(dog);
poodle.name = 'Spot';
poodle.speak();   // Spot barked!
```

One more thing, by deleting a property on an object, we can re-uncover the property from its prototype:

```
delete dog.speak;
poodle.speak();   // Spot made a noise.
```

Classes

A class declaration is a nice syntax for creating a prototype object in a way that looks like other class-based languages. Remember that JavaScript is not class-based, it is object-based; that is, inheritance is based on the object prototype, there are no real classes.

For more about declaring classes, see Class declaration on page 22.

© Chillisoft, 2018

Core Library

Here are some key JavaScript types that are used extensively. It's worth taking the time to explore their capabilities:

Array	A highly versatile type that has array access syntax [] but also operates like a list with dynamic size.	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
Map	A data structure that holds key-value pairs	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map
Math	An object that provides many mathematical functions and constants	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math
String	The String object	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

© Chillisoft, 2018

References & Further Reading

- *JavaScript: The Good Parts*, Douglas Crockford, © 2008 Yahoo! Inc. Published by O'Reilly Media.
- *Ecma-262 Edition 5.1, The ECMAScript Language Specification, © 2011 Ecma International*: https://www.ecma-international.org/ecma-262/5.1/
- *ECMA-262 6th Edition, The ECMAScript 2015 Language Specification, © 2015 Ecma International:* https://www.ecma-international.org/ecma-262/6.0/
- *Eloquent JavaScript, 3rd Edition,* Marijn Haverbeke, https://eloquentjavascript.net/
- *https://developer.mozilla.org/en-US/docs/Web/JavaScript: The Mozilla Developer Network (MDN) JavaScript reference documentation.*