Object initializer

Objects can be initialized using `new Object()`, `Object.create()`, or using the *literal* notation (*initializer* notation). An object initializer is a comma-delimited list of zero or more pairs of property names and associated values of an object, enclosed in curly braces (`{}`).

JavaScript Demo: Expressions - Object initializer

```
const object1 = {a: 'foo', b: 42, c: {}};

console.log(object1.a);
// expected output: "foo"

const a = 'foo';
const b = 42;
const c = {};
const object2 = {a: a, b: b, c: c};

console.log(object2.b);
// expected output: 42

const object3 = {a, b, c};

console.log(object3.a);
// expected output: "foo"
```

Run ›

Reset

Syntax

```
let o = {}
let o = {a: 'foo', b: 42, c: {}}

let a = 'foo', b = 42, c = {}
let o = {a: a, b: b, c: c}

let o = {
```

```
  property: function (parameters) {},
  get property() {},
  set property(value) {}
};
```

New notations in ECMAScript 2015

Please see the compatibility table for support for these notations. In non-supporting environments, these notations will lead to syntax errors.

```
// Shorthand property names (ES2015)
let a = 'foo', b = 42, c = {};
let o = {a, b, c}

// Shorthand method names (ES2015)
let o = {
  property(parameters) {}
}

// Computed property names (ES2015)
let prop = 'foo'
let o = {
  [prop]: 'hey',
  ['b' + 'ar']: 'there'
}
```

Description

An object initializer is an expression that describes the initialization of an `Object`. Objects consist of *properties*, which are used to describe an object. Values of object properties can either contain primitive data types or other objects.

Creating objects

An empty object with no properties can be created like this:

```
let object = {}
```

However, the advantage of the *literal* or *initializer* notation is, that you are able to quickly create objects with properties inside the curly braces. You simply notate a list of `key: value` pairs delimited by commas.

The following code creates an object with three properties and the keys are `"foo"`, `"age"` and `"baz"`. The values of these keys are a string `"bar"`, the number `42`, and another object.

```
let object = {
  foo: 'bar',
  age: 42,
  baz: {myProp: 12}
}
```

Accessing properties

Once you have created an object, you might want to read or change them. Object properties can be accessed by using the dot notation or the bracket notation. (See property accessors for detailed information.)

```
object.foo // "bar"
object['age'] // 42

object.foo = 'baz'
```

Property definitions

We have already learned how to notate properties using the initializer syntax. Oftentimes, there are variables in your code that you would like to put into an object. You will see code like this:

```
let a = 'foo',
    b = 42,
    c = {};

let o = {
  a: a,
```

```
  b: b,
  c: c
}
```

With ECMAScript 2015, there is a shorter notation available to achieve the same:

```
let a = 'foo',
    b = 42,
    c = {};

// Shorthand property names (ES2015)
let o = {a, b, c}

// In other words,
console.log((o.a === {a}.a)) // true
```

Duplicate property names

When using the same name for your properties, the second property will overwrite the first.

```
let a = {x: 1, x: 2}
console.log(a) // {x: 2}
```

In ECMAScript 5 strict mode code, duplicate property names were considered a `SyntaxError`. With the introduction of computed property names making duplication possible at runtime, ECMAScript 2015 has removed this restriction.

```
function haveES2015DuplicatePropertySemantics() {
  'use strict';
  try {
    ({prop: 1, prop: 2});

    // No error thrown, duplicate property names allowed in strict mod
    return true;
  } catch(e) {
    // Error thrown, duplicates prohibited in strict mode
    return false;
  }
}
```

Method definitions

A property of an object can also refer to a function or a getter or setter method.

```
let o = {
  property: function (parameters) {},
  get property() {},
  set property(value) {}
}
```

In ECMAScript 2015, a shorthand notation is available, so that the keyword "`function`" is no longer necessary.

```
// Shorthand method names (ES2015)
let o = {
  property(parameters) {},
}
```

In ECMAScript 2015, there is a way to concisely define properties whose values are generator functions:

```
let o = {
  *generator() {
    ...........
  }
};
```

Which is equivalent to this ES5-like notation (but note that ECMAScript 5 has no generators):

```
let o = {
  generator: function* () {
    ...........
  }
};
```

For more information and examples about methods, see method definitions.

Computed property names

Starting with ECMAScript 2015, the object initializer syntax also supports computed property names. That allows you to put an expression in brackets `[]`, that will be computed and used as the property name. This is reminiscent of the bracket notation of the property accessor syntax, which you may have used to read and set properties already.

Now you can use a similar syntax in object literals, too:

```
// Computed property names (ES2015)
let i = 0
let a = {
  ['foo' + ++i]: i,
  ['foo' + ++i]: i,
  ['foo' + ++i]: i
}

console.log(a.foo1) // 1
console.log(a.foo2) // 2
console.log(a.foo3) // 3

let param = 'size'
let config = {
  [param]: 12,
  ['mobile' + param.charAt(0).toUpperCase() + param.slice(1)]: 4
}

console.log(config) // {size: 12, mobileSize: 4}
```

Spread properties

The Rest/Spread Properties for ECMAScript proposal (stage 4) adds spread properties to object literals. It copies own enumerable properties from a provided object onto a new object.

Shallow-cloning (excluding `prototype`) or merging objects is now possible using a shorter syntax than `Object.assign()`.

```
let obj1 = { foo: 'bar', x: 42 }
let obj2 = { foo: 'baz', y: 13 }
```

```
3
4  let clonedObj = { ...obj1 }
5  // Object { foo: "bar", x: 42 }
6
7  let mergedObj = { ...obj1, ...obj2 }
8  // Object { foo: "baz", x: 42, y: 13 }
```

Note that `Object.assign()` triggers setters, whereas the spread operator doesn't!

Prototype mutation

A property definition of the form `__proto__: value` or `"__proto__": value` does not create a property with the name `__proto__`. Instead, if the provided value is an object or `null`, it changes the `[[Prototype]]` of the created object to that value. (If the value is not an object or `null`, the object is not changed.)

```
1  let obj1 = {}
2  assert(Object.getPrototypeOf(obj1) === Object.prototype)
3
4  let obj2 = {__proto__: null}
5  assert(Object.getPrototypeOf(obj2) === null)
6
7  let protoObj = {}
8  let obj3 = {'__proto__': protoObj}
9  assert(Object.getPrototypeOf(obj3) === protoObj)
10
11 let obj4 = {__proto__: 'not an object or null'}
12 assert(Object.getPrototypeOf(obj4) === Object.prototype)
13 assert(!obj4.hasOwnProperty('__proto__'))
```

Only a single prototype mutation is permitted in an object literal. Multiple prototype mutations are a syntax error.

Property definitions that do not use "colon" notation are not prototype mutations. They are property definitions that behave identically to similar definitions using any other name.

```
1  let __proto__ = 'variable'
2
3  let obj1 = {__proto__}
4  assert(Object.getPrototypeOf(obj1) === Object.prototype)
```

```
assert(obj1.hasOwnProperty('__proto__'))
assert(obj1.__proto__ === 'variable')

let obj2 = {__proto__() { return 'hello'; }}
assert(obj2.__proto__() === 'hello')

let obj3 = {['__prot' + 'o__']: 17}
assert(obj3.__proto__ === 17)
```

Object literal notation vs JSON

The object literal notation is not the same as the **J**ava**S**cript **O**bject **N**otation (JSON). Although they look similar, there are differences between them:

- JSON permits *only* property definition using `"property": value` syntax. The property name must be double-quoted, and the definition cannot be a shorthand.
- In JSON the values can only be strings, numbers, arrays, `true`, `false`, `null`, or another (JSON) object.
- A function value (see "Methods" above) can not be assigned to a value in JSON.
- Objects like `Date` will be a string after `JSON.parse()`.
- `JSON.parse()` will reject computed property names and an error will be thrown.

Specifications

Specification
ECMAScript (ECMA-262) The definition of 'Object Initializer' in that specification.

Browser compatibility

Object initializer	
Chrome	1
Edge	12
Firefox	1
IE	1
Opera	4
Safari	1
WebView Android	1
Chrome Android	18
Firefox Android	4
Opera Android	10.1
Safari iOS	1
Samsung Internet Android	1.0
nodejs	0.1.100

Computed property names	
Chrome	47
Edge	12
Firefox	34
IE	No
Opera	34
Safari	8
WebView Android	47
Chrome Android	47
Firefox Android	34
Opera Android	34
Safari iOS	8
Samsung Internet Android	5.0

nodejs	4.0.0

Shorthand method names	
Chrome	47
Edge	12
Firefox	34
IE	No
Opera	34
Safari	9
WebView Android	47
Chrome Android	47
Firefox Android	34
Opera Android	34
Safari iOS	9
Samsung Internet Android	5.0
nodejs	4.0.0

Shorthand property names	
Chrome	47
Edge	12
Firefox	33
IE	No
Opera	34
Safari	9
WebView Android	47
Chrome Android	47
Firefox Android	33
Opera Android	34
Safari iOS	9
Samsung Internet Android	5.0

nodejs	4.0.0

Spread properties

Chrome	60
Edge	79
Firefox	55
IE	No
Opera	47
Safari	11.1
WebView Android	60
Chrome Android	60
Firefox Android	55
Opera Android	44
Safari iOS	11.3
Samsung Internet Android	8.0
nodejs	8.3.0

What are we missing?

Full support

No support

Experimental. Expect behavior to change in the future.

See also

- Property accessors
- `get` / `set`
- Method definitions
- Lexical grammar

Last modified: Mar 13, 2020, by MDN contributors

Related Topics

JavaScript

Tutorials:

- Complete beginners
- JavaScript Guide
- Intermediate
- Advanced

References:

- Built-in objects
- Expressions & operators
  - Arithmetic operators
    - Array comprehensions
  - Assignment operators
  - Bitwise operators
  - Comma operator
  - Comparison operators
  - Conditional (ternary) operator
  - Destructuring assignment
    - Expression closures
  - Function expression
    - Generator comprehensions
  - Grouping operator
    - Legacy generator function expression
  - Logical operators
  - Nullish coalescing operator

- Object initializer
- Operator precedence
- Optional chaining
- Pipeline operator
- Property accessors
- Spread syntax
- async function expression
- await
- class expression
- delete operator
- function* expression
- in operator
- instanceof
- new operator
- new.target
- super
- this
- typeof
- void operator
- yield
- yield*

- ▸ Statements & declarations
- ▸ Functions
- ▸ Classes
- ▸ Errors
- ▸ Misc