

Objects: the basics

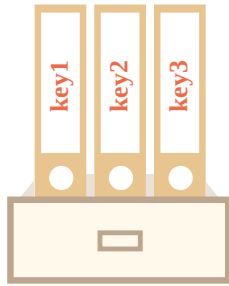
Objects

As we know from the chapter [Data types](#), there are seven data types in JavaScript. Six of them are called “primitive”, because their values contain only a single thing (be it a string or a number or whatever).

In contrast, objects are used to store keyed collections of various data and more complex entities. In JavaScript, objects penetrate almost every aspect of the language. So we must understand them first before going in-depth anywhere else.

An object can be created with figure brackets `{...}` with an optional list of *properties*. A property is a “key: value” pair, where `key` is a string (also called a “property name”), and `value` can be anything.

We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key. It's easy to find a file by its name or add/remove a file.



An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
let user = {}; // "object literal" syntax
```



Usually, the figure brackets `{ . . . }` are used. That declaration is called an *object literal*.

Literals and properties

We can immediately put some properties into `{ . . . }` as “key: value” pairs:

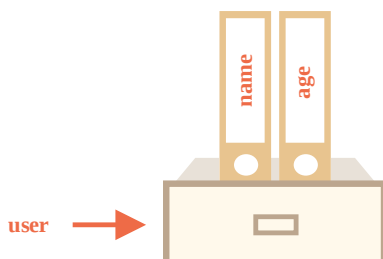
```
let user = { // an object
  name: "John", // by key "name" store value "John"
  age: 30 // by key "age" store value 30
};
```

A property has a key (also known as “name” or “identifier”) before the colon `:` and a value to the right of it.

In the `user` object, there are two properties:

1. The first property has the name `"name"` and the value `"John"`.
2. The second one has the name `"age"` and the value `30`.

The resulting `user` object can be imagined as a cabinet with two signed files labeled “name” and “age”.



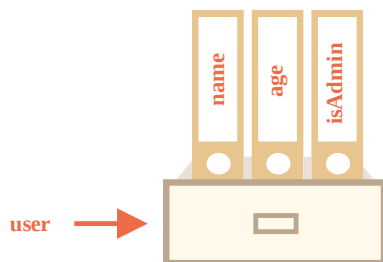
We can add, remove and read files from it any time.

Property values are accessible using the dot notation:

```
// get property values of the object:  
alert( user.name ); // John  
alert( user.age ); // 30
```

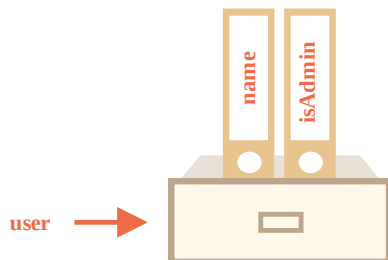
The value can be of any type. Let's add a boolean one:

```
user.isAdmin = true;
```



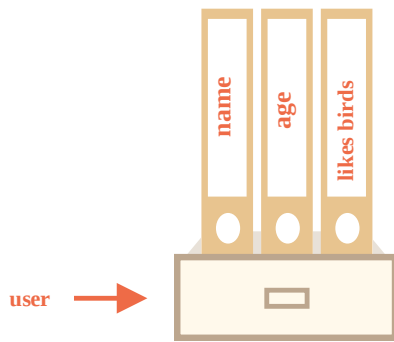
To remove a property, we can use `delete` operator:

```
delete user.age;
```



We can also use multiword property names, but then they must be quoted:

```
let user = {  
  name: "John",  
  age: 30,  
  "likes birds": true // multiword property name must be quoted  
};
```



The last property in the list may end with a comma:

```
let user = {  
  name: "John",  
  age: 30,  
}
```

That is called a “trailing” or “hanging” comma. Makes it easier to add/remove/move around properties, because all lines become alike.

Square brackets

For multiword properties, the dot access doesn't work:

```
// this would give a syntax error  
user.likes birds = true
```

JavaScript doesn't understand that. It thinks that we address `user.likes`, and then gives a syntax error when comes across unexpected `birds`.

The dot requires the key to be a valid variable identifier. That implies: contains no spaces, doesn't start with a digit and doesn't include special characters (`$` и `_` are allowed).

There's an alternative “square bracket notation” that works with any string:

```
let user = {};  
  
// set  
user["likes birds"] = true;  
  
// get  
alert(user["likes birds"]); // true  
  
// delete  
delete user["likes birds"];
```

Now everything is fine. Please note that the string inside the brackets is properly quoted (any type of quotes will do).

Square brackets also provide a way to obtain the property name as the result of any expression – as opposed to a literal string – like from a variable as follows:

```
let key = "likes birds";

// same as user["likes birds"] = true;
user[key] = true;
```

Here, the variable `key` may be calculated at run-time or depend on the user input. And then we use it to access the property. That gives us a great deal of flexibility.

For instance:

```
let user = {
  name: "John",
  age: 30
};

let key = prompt("What do you want to know about the user?", "name");

// access by variable
alert( user[key] ); // John (if enter "name")
```

The dot notation cannot be used in a similar way:

```
let user = {
  name: "John",
  age: 30
};

let key = "name";
alert( user.key ) // undefined
```

Computed properties

We can use square brackets in an object literal. That's called *computed properties*.

For instance:

```
let fruit = prompt("Which fruit to buy?", "apple");

let bag = {
  [fruit]: 5, // the name of the property is taken from the variable fruit
};

alert( bag.apple ); // 5 if fruit="apple"
```

The meaning of a computed property is simple: `[fruit]` means that the property name should be taken from `fruit`.

So, if a visitor enters `"apple"`, `bag` will become `{apple: 5}`.

Essentially, that works the same as:

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {};

// take property name from the fruit variable
bag[fruit] = 5;
```

...But looks nicer.

We can use more complex expressions inside square brackets:

```
let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

Square brackets are much more powerful than the dot notation. They allow any property names and variables. But they are also more cumbersome to write.

So most of the time, when property names are known and simple, the dot is used. And if we need something more complex, then we switch to square brackets.

Property value shorthand

In real code we often use existing variables as values for property names.

For instance:

```
function makeUser(name, age) {
  return {
    name: name,
    age: age
    // ...other properties
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

In the example above, properties have the same names as variables. The use-case of making a property from a variable is so common, that there's a special *property value shorthand* to make it shorter.

Instead of `name : name` we can just write `name` , like this:

```
function makeUser(name, age) {
  return {
    name, // same as name: name
    age  // same as age: age
    // ...
  };
}
```

```
};  
}
```

We can use both normal properties and shorthands in the same object:

```
let user = {  
  name, // same as name:name  
  age: 30  
};
```

Property names limitations

Property names (keys) must be either strings or symbols (a special type for identifiers, to be covered later).

Other types are automatically converted to strings.

For instance, a number `0` becomes a string `"0"` when used as a property key:

```
let obj = {  
  0: "test" // same as "0": "test"  
};  
  
// both alerts access the same property (the number 0 is converted to string "0")  
alert( obj["0"] ); // test  
alert( obj[0] ); // test (same property)
```

Reserved words are allowed as property names.

As we already know, a variable cannot have a name equal to one of language-reserved words like `for`, `let`, `return` etc.

But for an object property, there's no such restriction. Any name is fine:

```
let obj = {  
  for: 1,  
  let: 2,  
  return: 3  
};  
  
alert( obj.for + obj.let + obj.return ); // 6
```

We can use any string as a key, but there's a special property named `__proto__` that gets special treatment for historical reasons.

For instance, we can't set it to a non-object value:

```
let obj = {};  
obj.__proto__ = 5; // assign a number  
alert(obj.__proto__); // [object Object] - the value is an object, didn't work as intended
```

As we see from the code, the assignment to a primitive `5` is ignored.

The nature of `__proto__` will be revealed in detail later in the chapter [Prototypal inheritance](#).

As for now, it's important to know that such behavior of `__proto__` can become a source of bugs and even vulnerabilities if we intend to store user-provided keys in an object.

The problem is that a visitor may choose `__proto__` as the key, and the assignment logic will be ruined (as shown above).

Later we'll see workarounds for the problem:

1. We'll see how to make an objects treat `__proto__` as a regular property in the chapter [Prototype methods, objects without __proto__](#).
2. There's also study another data structure [Map](#) in the chapter [Map and Set](#), which supports arbitrary keys.

Property existence test, "in" operator

A notable objects feature is that it's possible to access any property. There will be no error if the property doesn't exist! Accessing a non-existing property just returns `undefined`. It provides a very common way to test whether the property exists – to get it and compare vs `undefined`:

```
let user = {};  
  
alert( user.noSuchProperty === undefined ); // true means "no such property"
```

There also exists a special operator `"in"` to check for the existence of a property.

The syntax is:

```
"key" in object
```

For instance:

```
let user = { name: "John", age: 30 };  
  
alert( "age" in user ); // true, user.age exists  
alert( "blabla" in user ); // false, user.blabla doesn't exist
```

Please note that on the left side of `in` there must be a *property name*. That's usually a quoted string.

If we omit quotes, that would mean a variable containing the actual name will be tested. For instance:

```
let user = { age: 30 };  
  
let key = "age";  
alert( key in user ); // true, takes the name from key and checks for such property
```


Using “in” for properties that store undefined

Usually, the strict comparison `=== undefined` check the property existence just fine. But there's a special case when it fails, but `"in"` works correctly.

It's when an object property exists, but stores `undefined` :

```
let obj = {  
  test: undefined  
};  
  
alert( obj.test ); // it's undefined, so - no such property?  
  
alert( "test" in obj ); // true, the property does exist!
```

In the code above, the property `obj.test` technically exists. So the `in` operator works right.

Situations like this happen very rarely, because `undefined` is usually not assigned. We mostly use `null` for “unknown” or “empty” values. So the `in` operator is an exotic guest in the code.

The “for...in” loop

To walk over all keys of an object, there exists a special form of the loop: `for...in`. This is a completely different thing from the `for(;;)` construct that we studied before.

The syntax:

```
for (key in object) {  
  // executes the body for each key among object properties  
}
```

For instance, let's output all properties of `user` :

```
let user = {  
  name: "John",  
  age: 30,  
  isAdmin: true  
};  
  
for (let key in user) {  
  // keys  
  alert( key ); // name, age, isAdmin  
  // values for the keys  
  alert( user[key] ); // John, 30, true  
}
```

Note that all “for” constructs allow us to declare the looping variable inside the loop, like `let key` here.

Also, we could use another variable name here instead of `key`. For instance, `"for (let prop in obj)"` is also widely used.

Ordered like an object

Are objects ordered? In other words, if we loop over an object, do we get all properties in the same order they were added? Can we rely on this?

The short answer is: “ordered in a special fashion”: integer properties are sorted, others appear in creation order. The details follow.

As an example, let’s consider an object with the phone codes:

```
let codes = {  
  "49": "Germany",  
  "41": "Switzerland",  
  "44": "Great Britain",  
  // ..,  
  "1": "USA"  
};  
  
for (let code in codes) {  
  alert(code); // 1, 41, 44, 49  
}
```

The object may be used to suggest a list of options to the user. If we’re making a site mainly for German audience then we probably want `49` to be the first.

But if we run the code, we see a totally different picture:

- USA (1) goes first
- then Switzerland (41) and so on.

The phone codes go in the ascending sorted order, because they are integers. So we see `1`, `41`, `44`, `49`.

Integer properties? What’s that?

The “integer property” term here means a string that can be converted to-and-from an integer without a change.

So, “49” is an integer property name, because when it’s transformed to an integer number and back, it’s still the same. But “+49” and “1.2” are not:

```
// Math.trunc is a built-in function that removes the decimal part  
alert( String(Math.trunc(Number("49"))) ); // "49", same, integer property  
alert( String(Math.trunc(Number("+49"))) ); // "49", not same "+49" ⇒ not integer property  
alert( String(Math.trunc(Number("1.2"))) ); // "1", not same "1.2" ⇒ not integer property
```

...On the other hand, if the keys are non-integer, then they are listed in the creation order, for instance:

```
let user = {
  name: "John",
  surname: "Smith"
};
user.age = 25; // add one more

// non-integer properties are listed in the creation order
for (let prop in user) {
  alert( prop ); // name, surname, age
}
```

So, to fix the issue with the phone codes, we can “cheat” by making the codes non-integer. Adding a plus “+” sign before each code is enough.

Like this:

```
let codes = {
  "+49": "Germany",
  "+41": "Switzerland",
  "+44": "Great Britain",
  // ..,
  "+1": "USA"
};

for (let code in codes) {
  alert( +code ); // 49, 41, 44, 1
}
```

Now it works as intended.

Copying by reference

One of the fundamental differences of objects vs primitives is that they are stored and copied “by reference”.

Primitive values: strings, numbers, booleans – are assigned/copied “as a whole value”.

For instance:

```
let message = "Hello!";
let phrase = message;
```

As a result we have two independent variables, each one is storing the string “Hello!”.

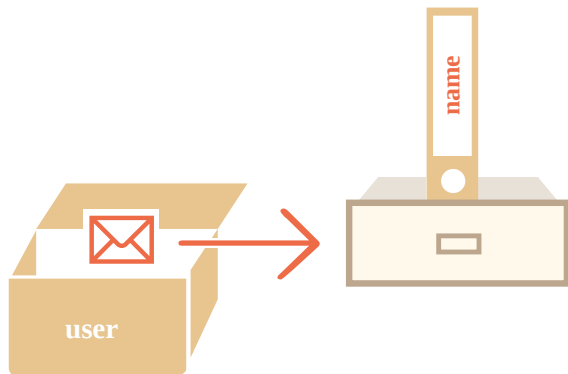


Objects are not like that.

A variable stores not the object itself, but its “address in memory”, in other words “a reference” to it.

Here’s the picture for the object:

```
let user = {  
  name: "John"  
};
```



Here, the object is stored somewhere in memory. And the variable `user` has a “reference” to it.

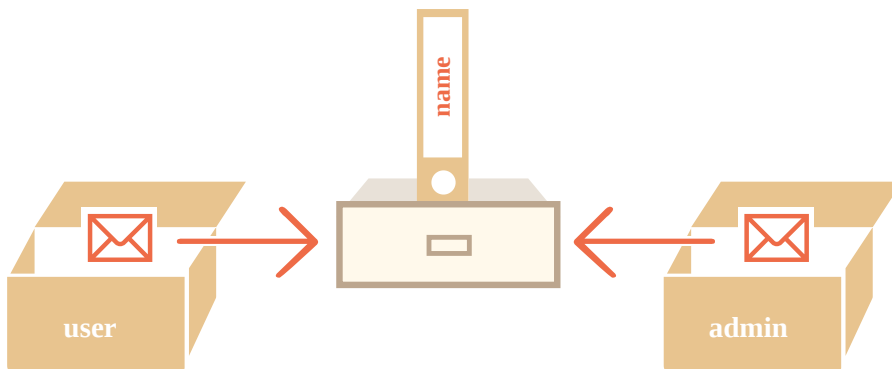
When an object variable is copied – the reference is copied, the object is not duplicated.

If we imagine an object as a cabinet, then a variable is a key to it. Copying a variable duplicates the key, but not the cabinet itself.

For instance:

```
let user = { name: "John" };  
  
let admin = user; // copy the reference
```

Now we have two variables, each one with the reference to the same object:



We can use any variable to access the cabinet and modify its contents:

```
let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // changed by the "admin" reference

alert(user.name); // 'Pete', changes are seen from the "user" reference
```

The example above demonstrates that there is only one object. As if we had a cabinet with two keys and used one of them (`admin`) to get into it. Then, if we later use the other key (`user`) we would see changes.

Comparison by reference

The equality `==` and strict equality `===` operators for objects work exactly the same.

Two objects are equal only if they are the same object.

For instance, if two variables reference the same object, they are equal:

```
let a = {};
let b = a; // copy the reference

alert( a == b ); // true, both variables reference the same object
alert( a === b ); // true
```

And here two independent objects are not equal, even though both are empty:

```
let a = {};
let b = {}; // two independent objects

alert( a == b ); // false
```

For comparisons like `obj1 > obj2` or for a comparison against a primitive `obj == 5`, objects are converted to primitives. We'll study how object conversions work very soon, but to tell the truth, such comparisons are necessary very rarely and usually are a result of a coding mistake.

Const object

An object declared as `const` *can* be changed.

For instance:

```
const user = {
  name: "John"
};

user.age = 25; // (*)

alert(user.age); // 25
```

It might seem that the line `(*)` would cause an error, but no, there's totally no problem. That's because `const` fixes only value of `user` itself. And here `user` stores the reference to the same object all the time. The line `(*)` goes *inside* the object, it doesn't reassign `user`.

The `const` would give an error if we try to set `user` to something else, for instance:

```
const user = {
  name: "John"
};

// Error (can't reassign user)
user = {
  name: "Pete"
};
```

...But what if we want to make constant object properties? So that `user.age = 25` would give an error. That's possible too. We'll cover it in the chapter [Property flags and descriptors](#).

Cloning and merging, `Object.assign`

So, copying an object variable creates one more reference to the same object.

But what if we need to duplicate an object? Create an independent copy, a clone?

That's also doable, but a little bit more difficult, because there's no built-in method for that in JavaScript. Actually, that's rarely needed. Copying by reference is good most of the time.

But if we really want that, then we need to create a new object and replicate the structure of the existing one by iterating over its properties and copying them on the primitive level.

Like this:


```
let user = {
  name: "John",
  age: 30
};

let clone = {}; // the new empty object

// let's copy all user properties into it
for (let key in user) {
  clone[key] = user[key];
}

// now clone is a fully independent clone
clone.name = "Pete"; // changed the data in it

alert( user.name ); // still John in the original object
```

Also we can use the method [Object.assign](#)  for that.

The syntax is:

```
Object.assign(dest, [src1, src2, src3...])
```

- Arguments `dest`, and `src1, ..., srcN` (can be as many as needed) are objects.
- It copies the properties of all objects `src1, ..., srcN` into `dest`. In other words, properties of all arguments starting from the 2nd are copied into the 1st. Then it returns `dest`.

For instance, we can use it to merge several objects into one:

```
let user = { name: "John" };

let permissions1 = { canView: true };
let permissions2 = { canEdit: true };

// copies all properties from permissions1 and permissions2 into user
Object.assign(user, permissions1, permissions2);

// now user = { name: "John", canView: true, canEdit: true }
```

If the receiving object (`user`) already has the same named property, it will be overwritten:

```
let user = { name: "John" };

// overwrite name, add isAdmin
Object.assign(user, { name: "Pete", isAdmin: true });

// now user = { name: "Pete", isAdmin: true }
```

We also can use `Object.assign` to replace the loop for simple cloning:

```
let user = {
  name: "John",
  age: 30
};

let clone = Object.assign({}, user);
```

It copies all properties of `user` into the empty object and returns it. Actually, the same as the loop, but shorter.

Until now we assumed that all properties of `user` are primitive. But properties can be references to other objects. What to do with them?

Like this:

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};
```

```
alert( user.sizes.height ); // 182
```

Now it's not enough to copy `clone.sizes = user.sizes`, because the `user.sizes` is an object, it will be copied by reference. So `clone` and `user` will share the same sizes:

Like this:

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = Object.assign({}, user);

alert( user.sizes === clone.sizes ); // true, same object

// user and clone share sizes
user.sizes.width++; // change a property from one place
alert(clone.sizes.width); // 51, see the result from the other one
```

To fix that, we should use the cloning loop that examines each value of `user[key]` and, if it's an object, then replicate its structure as well. That is called a “deep cloning”.

There's a standard algorithm for deep cloning that handles the case above and more complex cases, called the [Structured cloning algorithm](#). In order not to reinvent the wheel, we can use a working implementation of it from the JavaScript library [lodash](#), the method is called [_.cloneDeep\(obj\)](#).

Summary

Objects are associative arrays with several special features.

They store properties (key-value pairs), where:

- Property keys must be strings or symbols (usually strings).
- Values can be of any type.

To access a property, we can use:

- The dot notation: `obj.property`.
- Square brackets notation `obj["property"]`. Square brackets allow to take the key from a variable, like `obj[varWithKey]`.

Additional operators:

- To delete a property: `delete obj.prop`.
- To check if a property with the given key exists: `"key" in obj`.
- To iterate over an object: `for (let key in obj) loop`.

Objects are assigned and copied by reference. In other words, a variable stores not the “object value”, but a “reference” (address in memory) for the value. So copying such a variable or passing it as a function argument copies that reference, not the object. All operations via copied references (like adding/removing properties) are performed on the same single object.

To make a “real copy” (a clone) we can use `Object.assign` or `_.cloneDeep(obj)` [↗](#).

What we’ve studied in this chapter is called a “plain object”, or just `Object`.

There are many other kinds of objects in JavaScript:

- `Array` to store ordered data collections,
- `Date` to store the information about the date and time,
- `Error` to store the information about an error.
- ...And so on.

They have their special features that we’ll study later. Sometimes people say something like “Array type” or “Date type”, but formally they are not types of their own, but belong to a single “object” data type. And they extend it in various ways.

Objects in JavaScript are very powerful. Here we’ve just scratched the surface of a topic that is really huge. We’ll be closely working with objects and learning more about them in further parts of the tutorial.

Garbage collection

Memory management in JavaScript is performed automatically and invisibly to us. We create primitives, objects, functions... All that takes memory.

What happens when something is not needed any more? How does the JavaScript engine discover it and clean it up?

Reachability

The main concept of memory management in JavaScript is *reachability*.

Simply put, “reachable” values are those that are accessible or usable somehow. They are guaranteed to be stored in memory.

1. There’s a base set of inherently reachable values, that cannot be deleted for obvious reasons.

For instance:

- Local variables and parameters of the current function.
- Variables and parameters for other functions on the current chain of nested calls.
- Global variables.
- (there are some other, internal ones as well)

These values are called *roots*.

2. Any other value is considered reachable if it’s reachable from a root by a reference or by a chain of references.

For instance, if there’s an object in a local variable, and that object has a property referencing another object, that object is considered reachable. And those that it references are also

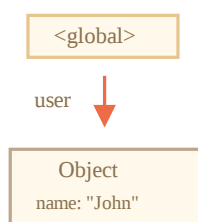
reachable. Detailed examples to follow.

There's a background process in the JavaScript engine that is called [garbage collector](#) . It monitors all objects and removes those that have become unreachable.

A simple example

Here's the simplest example:

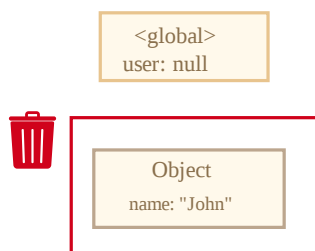
```
// user has a reference to the object
let user = {
  name: "John"
};
```



Here the arrow depicts an object reference. The global variable "user" references the object {name: "John"} (we'll call it John for brevity). The "name" property of John stores a primitive, so it's painted inside the object.

If the value of user is overwritten, the reference is lost:

```
user = null;
```



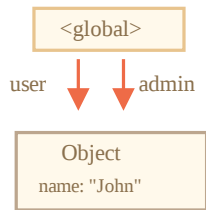
Now John becomes unreachable. There's no way to access it, no references to it. Garbage collector will junk the data and free the memory.

Two references

Now let's imagine we copied the reference from user to admin :

```
// user has a reference to the object
let user = {
  name: "John"
};
```

```
let admin = user;
```



Now if we do the same:

```
user = null;
```

...Then the object is still reachable via `admin` global variable, so it's in memory. If we overwrite `admin` too, then it can be removed.

Interlinked objects

Now a more complex example. The family:

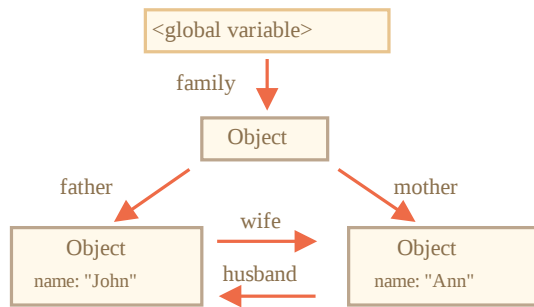
```
function marry(man, woman) {
  woman.husband = man;
  man.wife = woman;

  return {
    father: man,
    mother: woman
  }
}

let family = marry({
  name: "John"
}, {
  name: "Ann"
});
```

Function `marry` “marries” two objects by giving them references to each other and returns a new object that contains them both.

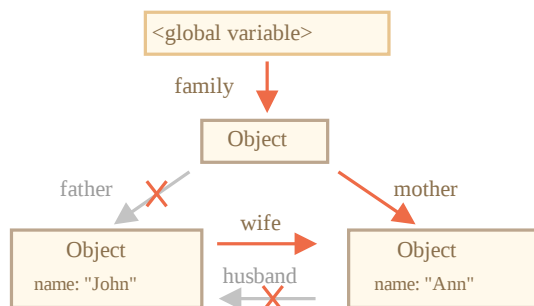
The resulting memory structure:



As of now, all objects are reachable.

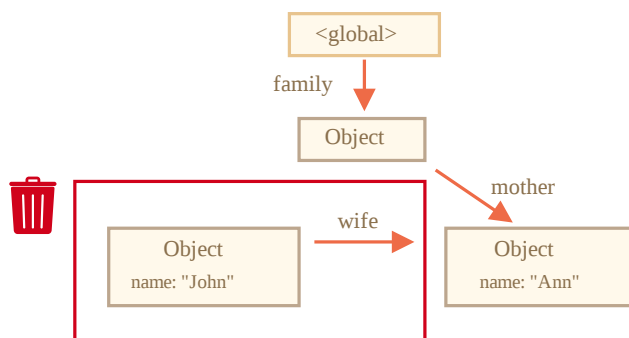
Now let's remove two references:

```
delete family.father;
delete family.mother.husband;
```



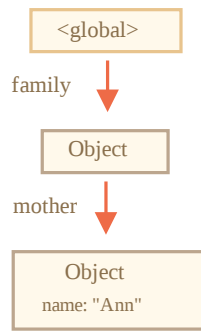
It's not enough to delete only one of these two references, because all objects would still be reachable.

But if we delete both, then we can see that John has no incoming reference any more:



Outgoing references do not matter. Only incoming ones can make an object reachable. So, John is now unreachable and will be removed from the memory with all its data that also became inaccessible.

After garbage collection:



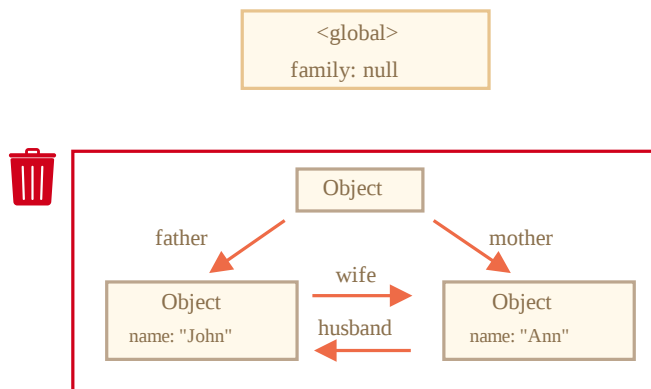
Unreachable island

It is possible that the whole island of interlinked objects becomes unreachable and is removed from the memory.

The source object is the same as above. Then:

```
family = null;
```

The in-memory picture becomes:



This example demonstrates how important the concept of reachability is.

It's obvious that John and Ann are still linked, both have incoming references. But that's not enough.

The former "family" object has been unlinked from the root, there's no reference to it any more, so the whole island becomes unreachable and will be removed.

Internal algorithms

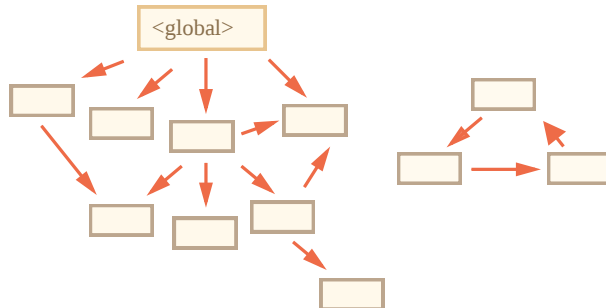
The basic garbage collection algorithm is called "mark-and-sweep".

The following "garbage collection" steps are regularly performed:

- The garbage collector takes roots and "marks" (remembers) them.
- Then it visits and "marks" all references from them.

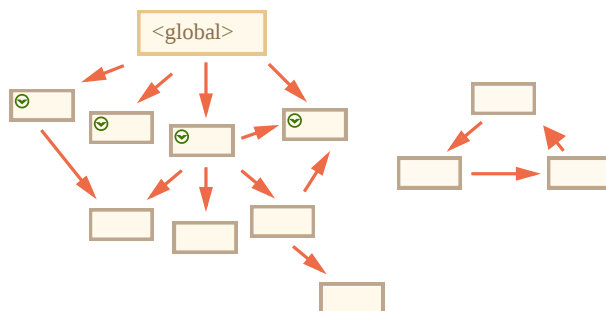
- Then it visits marked objects and marks *their* references. All visited objects are remembered, so as not to visit the same object twice in the future.
- ...And so on until every reachable (from the roots) references are visited.
- All objects except marked ones are removed.

For instance, let our object structure look like this:

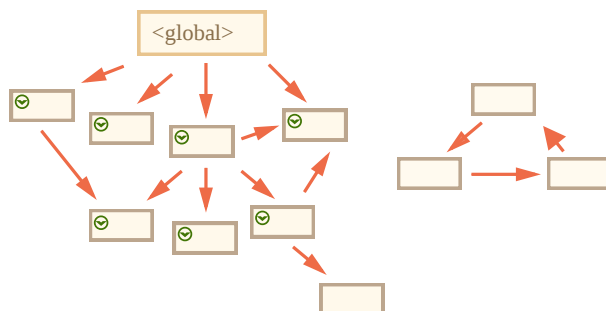


We can clearly see an “unreachable island” to the right side. Now let’s see how “mark-and-sweep” garbage collector deals with it.

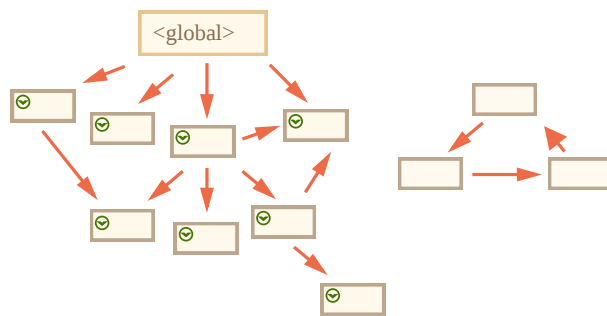
The first step marks the roots:



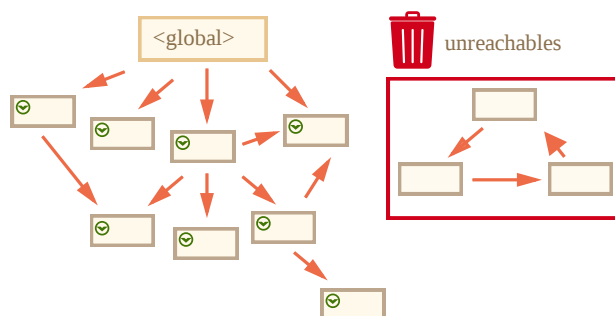
Then their references are marked:



...And their references, while possible:



Now the objects that could not be visited in the process are considered unreachable and will be removed:



We can also imagine the process as spilling a huge bucket of paint from the roots, that flows through all references and marks all reachable objects. The unmarked ones are then removed.

That's the concept of how garbage collection works. JavaScript engines apply many optimizations to make it run faster and not affect the execution.

Some of the optimizations:

- **Generational collection** – objects are split into two sets: “new ones” and “old ones”. Many objects appear, do their job and die fast, they can be cleaned up aggressively. Those that survive for long enough, become “old” and are examined less often.
- **Incremental collection** – if there are many objects, and we try to walk and mark the whole object set at once, it may take some time and introduce visible delays in the execution. So the engine tries to split the garbage collection into pieces. Then the pieces are executed one by one, separately. That requires some extra bookkeeping between them to track changes, but we have many tiny delays instead of a big one.
- **Idle-time collection** – the garbage collector tries to run only while the CPU is idle, to reduce the possible effect on the execution.

There exist other optimizations and flavours of garbage collection algorithms. As much as I'd like to describe them here, I have to hold off, because different engines implement different tweaks and techniques. And, what's even more important, things change as engines develop, so studying deeper “in advance”, without a real need is probably not worth that. Unless, of course, it is a matter of pure interest, then there will be some links for you below.

Summary

The main things to know:

- Garbage collection is performed automatically. We cannot force or prevent it.

- Objects are retained in memory while they are reachable.
- Being referenced is not the same as being reachable (from a root): a pack of interlinked objects can become unreachable as a whole.

Modern engines implement advanced algorithms of garbage collection.

A general book “The Garbage Collection Handbook: The Art of Automatic Memory Management” (R. Jones et al) covers some of them.

If you are familiar with low-level programming, the more detailed information about V8 garbage collector is in the article [A tour of V8: Garbage Collection](#) .

[V8 blog](#) also publishes articles about changes in memory management from time to time.

Naturally, to learn the garbage collection, you’d better prepare by learning about V8 internals in general and read the blog of [Vyacheslav Egorov](#) who worked as one of V8 engineers. I’m saying: “V8”, because it is best covered with articles in the internet. For other engines, many approaches are similar, but garbage collection differs in many aspects.

In-depth knowledge of engines is good when you need low-level optimizations. It would be wise to plan that as the next step after you’re familiar with the language.

Symbol type

By specification, object property keys may be either of string type, or of symbol type. Not numbers, not booleans, only strings or symbols, these two types.

Till now we’ve been using only strings. Now let’s see the benefits that symbols can give us.

Symbols

A “symbol” represents a unique identifier.

A value of this type can be created using `Symbol()` :

```
// id is a new symbol
let id = Symbol();
```

Upon creation, we can give symbol a description (also called a symbol name), mostly useful for debugging purposes:

```
// id is a symbol with the description "id"
let id = Symbol("id");
```

Symbols are guaranteed to be unique. Even if we create many symbols with the same description, they are different values. The description is just a label that doesn’t affect anything.

For instance, here are two symbols with the same description – they are not equal:

```
let id1 = Symbol("id");
let id2 = Symbol("id");
```



```
alert(id1 == id2); // false
```

If you are familiar with Ruby or another language that also has some sort of “symbols” – please don’t be misguided. JavaScript symbols are different.

⚠ Symbols don’t auto-convert to a string

Most values in JavaScript support implicit conversion to a string. For instance, we can `alert` almost any value, and it will work. Symbols are special. They don’t auto-convert.

For instance, this `alert` will show an error:

```
let id = Symbol("id");
alert(id); // TypeError: Cannot convert a Symbol value to a string
```

That’s a “language guard” against messing up, because strings and symbols are fundamentally different and should not accidentally convert one into another.

If we really want to show a symbol, we need to explicitly call `.toString()` on it, like here:

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id), now it works
```

Or get `symbol.description` property to show the description only:

```
let id = Symbol("id");
alert(id.description); // id
```

“Hidden” properties

Symbols allow us to create “hidden” properties of an object, that no other part of code can accidentally access or overwrite.

For instance, if we’re working with `user` objects, that belong to a third-party code. We’d like to add identifiers to them.

Let’s use a symbol key for it:

```
let user = { // belongs to another code
  name: "John"
};

let id = Symbol("id");

user[id] = 1;

alert( user[id] ); // we can access the data using the symbol as the key
```

What's the benefit of using `Symbol("id")` over a string `"id"` ?

As `user` objects belongs to another code, and that code also works with them, we shouldn't just add any fields to it. That's unsafe. But a symbol cannot be accessed accidentally, the third-party code probably won't even see it, so it's probably all right to do.

Also, imagine that another script wants to have its own identifier inside `user` , for its own purposes. That may be another JavaScript library, so that the scripts are completely unaware of each other.

Then that script can create its own `Symbol("id")` , like this:

```
// ...
let id = Symbol("id");

user[id] = "Their id value";
```

There will be no conflict between our and their identifiers, because symbols are always different, even if they have the same name.

...But if we used a string `"id"` instead of a symbol for the same purpose, then there *would* be a conflict:

```
let user = { name: "John" };

// Our script uses "id" property
user.id = "Our id value";

// ...Another script also wants "id" for its purposes...

user.id = "Their id value"
// Boom! overwritten by another script!
```

Symbols in a literal

If we want to use a symbol in an object literal `{...}` , we need square brackets around it.

Like this:

```
let id = Symbol("id");

let user = {
  name: "John",
  [id]: 123 // not "id: 123"
};
```

That's because we need the value from the variable `id` as the key, not the string `"id"`.

Symbols are skipped by for...in

Symbolic properties do not participate in `for...in` loop.

For instance:

```

let id = Symbol("id");
let user = {
  name: "John",
  age: 30,
  [id]: 123
};

for (let key in user) alert(key); // name, age (no symbols)

// the direct access by the symbol works
alert( "Direct: " + user[id] );

```

`Object.keys(user)` also ignores them. That's a part of the general “hiding symbolic properties” principle. If another script or a library loops over our object, it won't unexpectedly access a symbolic property.

In contrast, `Object.assign` [↗](#) copies both string and symbol properties:

```

let id = Symbol("id");
let user = {
  [id]: 123
};

let clone = Object.assign({}, user);

alert( clone[id] ); // 123

```

There's no paradox here. That's by design. The idea is that when we clone an object or merge objects, we usually want *all* properties to be copied (including symbols like `id`).

Global symbols

As we've seen, usually all symbols are different, even if they have the same name. But sometimes we want same-named symbols to be same entities. For instance, different parts of our application want to access symbol `"id"` meaning exactly the same property.

To achieve that, there exists a *global symbol registry*. We can create symbols in it and access them later, and it guarantees that repeated accesses by the same name return exactly the same symbol.

In order to read (create if absent) a symbol from the registry, use `Symbol.for(key)`.

That call checks the global registry, and if there's a symbol described as `key`, then returns it, otherwise creates a new symbol `Symbol(key)` and stores it in the registry by the given `key`.

For instance:

```

// read from the global registry
let id = Symbol.for("id"); // if the symbol did not exist, it is created

// read it again (maybe from another part of the code)
let idAgain = Symbol.for("id");

```

```
// the same symbol
alert( id === idAgain ); // true
```

Symbols inside the registry are called *global symbols*. If we want an application-wide symbol, accessible everywhere in the code – that’s what they are for.

That sounds like Ruby

In some programming languages, like Ruby, there’s a single symbol per name.

In JavaScript, as we can see, that’s right for global symbols.

Symbol.keyFor

For global symbols, not only `Symbol.for(key)` returns a symbol by name, but there’s a reverse call: `Symbol.keyFor(sym)`, that does the reverse: returns a name by a global symbol.

For instance:

```
// get symbol by name
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");

// get name by symbol
alert( Symbol.keyFor(sym) ); // name
alert( Symbol.keyFor(sym2) ); // id
```

The `Symbol.keyFor` internally uses the global symbol registry to look up the key for the symbol. So it doesn’t work for non-global symbols. If the symbol is not global, it won’t be able to find it and return `undefined`.

That said, any symbols have `description` property.

For instance:


```
let globalSymbol = Symbol.for("name");
let localSymbol = Symbol("name");

alert( Symbol.keyFor(globalSymbol) ); // name, global symbol
alert( Symbol.keyFor(localSymbol) ); // undefined, not global

alert( localSymbol.description ); // name
```

System symbols

There exist many “system” symbols that JavaScript uses internally, and we can use them to fine-tune various aspects of our objects.

They are listed in the specification in the [Well-known symbols](#)  table:

- `Symbol.hasInstance`
- `Symbol.isConcatSpreadable`

- `Symbol.iterator`
- `Symbol.toPrimitive`
- ...and so on.

For instance, `Symbol.toPrimitive` allows us to describe object to primitive conversion. We'll see its use very soon.

Other symbols will also become familiar when we study the corresponding language features.

Summary

`Symbol` is a primitive type for unique identifiers.

Symbols are created with `Symbol()` call with an optional description (name).

Symbols are always different values, even if they have the same name. If we want same-named symbols to be equal, then we should use the global registry: `Symbol.for(key)` returns (creates if needed) a global symbol with `key` as the name. Multiple calls of `Symbol.for` with the same `key` return exactly the same symbol.

Symbols have two main use cases:

1. "Hidden" object properties. If we want to add a property into an object that "belongs" to another script or a library, we can create a symbol and use it as a property key. A symbolic property does not appear in `for...in`, so it won't be accidentally processed together with other properties. Also it won't be accessed directly, because another script does not have our symbol. So the property will be protected from accidental use or overwrite.

So we can "covertly" hide something into objects that we need, but others should not see, using symbolic properties.

2. There are many system symbols used by JavaScript which are accessible as `Symbol.*`. We can use them to alter some built-in behaviors. For instance, later in the tutorial we'll use `Symbol.iterator` for [iterables](#), `Symbol.toPrimitive` to setup [object-to-primitive conversion](#) and so on.

Technically, symbols are not 100% hidden. There is a built-in method [Object.getOwnPropertySymbols\(obj\)](#) that allows us to get all symbols. Also there is a method named [Reflect.ownKeys\(obj\)](#) that returns *all* keys of an object including symbolic ones. So they are not really hidden. But most libraries, built-in functions and syntax constructs don't use these methods.

Object methods, "this"

Objects are usually created to represent entities of the real world, like users, orders and so on:

```
let user = {
  name: "John",
  age: 30
};
```

And, in the real world, a user can *act*: select something from the shopping cart, login, logout etc.

Actions are represented in JavaScript by functions in properties.

Method examples

For a start, let's teach the `user` to say hello:

```
let user = {  
  name: "John",  
  age: 30  
};  
  
user.sayHi = function() {  
  alert("Hello!");  
};  
  
user.sayHi(); // Hello!
```

Here we've just used a Function Expression to create the function and assign it to the property `user.sayHi` of the object.

Then we can call it. The user can now speak!


A function that is the property of an object is called its *method*.

So, here we've got a method `sayHi` of the object `user`.

Of course, we could use a pre-declared function as a method, like this:

```
let user = {  
  // ...  
};  
  
// first, declare  
function sayHi() {  
  alert("Hello!");  
};  
  
// then add as a method  
user.sayHi = sayHi;  
  
user.sayHi(); // Hello!
```

Object-oriented programming

When we write our code using objects to represent entities, that's called [object-oriented programming](#) , in short: "OOP".

OOP is a big thing, an interesting science of its own. How to choose the right entities? How to organize the interaction between them? That's architecture, and there are great books on that topic, like "Design Patterns: Elements of Reusable Object-Oriented Software" by E.Gamma, R.Helm, R.Johnson, J.Vissides or "Object-Oriented Analysis and Design with Applications" by G.Booch, and more.

Method shorthand

There exists a shorter syntax for methods in an object literal:

```
// these objects do the same

user = {
  sayHi: function() {
    alert("Hello");
  }
};

// method shorthand looks better, right?
user = {
  sayHi() { // same as "sayHi: function()"
    alert("Hello");
  }
};
```

As demonstrated, we can omit `"function"` and just write `sayHi()`.

To tell the truth, the notations are not fully identical. There are subtle differences related to object inheritance (to be covered later), but for now they do not matter. In almost all cases the shorter syntax is preferred.

“this” in methods

It's common that an object method needs to access the information stored in the object to do its job.

For instance, the code inside `user.sayHi()` may need the name of the `user`.

To access the object, a method can use the `this` keyword.

The value of `this` is the object “before dot”, the one used to call the method.

For instance:

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    // "this" is the "current object"
    alert(this.name);
  }
};

user.sayHi(); // John
```

Here during the execution of `user.sayHi()`, the value of `this` will be `user`.

Technically, it's also possible to access the object without `this`, by referencing it via the outer variable:

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    alert(user.name); // "user" instead of "this"
  }
};
```

...But such code is unreliable. If we decide to copy `user` to another variable, e.g. `admin = user` and overwrite `user` with something else, then it will access the wrong object.

That's demonstrated below:

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    alert( user.name ); // leads to an error
  }
};

let admin = user;
user = null; // overwrite to make things obvious

admin.sayHi(); // Whoops! inside sayHi(), the old name is used! error!
```

If we used `this.name` instead of `user.name` inside the `alert`, then the code would work.

“this” is not bound

In JavaScript, keyword `this` behaves unlike most other programming languages. It can be used in any function.

There's no syntax error in the following example:

```
function sayHi() {
  alert( this.name );
}
```

The value of `this` is evaluated during the run-time, depending on the context.

For instance, here the same function is assigned to two different objects and has different “this” in the calls:

```
let user = { name: "John" };
let admin = { name: "Admin" };
```



```
function sayHi() {
  alert( this.name );
}

// use the same function in two objects
user.f = sayHi;
admin.f = sayHi;

// these calls have different this
// "this" inside the function is the object "before the dot"
user.f(); // John (this == user)
admin.f(); // Admin (this == admin)

admin['f'](); // Admin (dot or square brackets access the method - doesn't matter)
```

The rule is simple: if `obj.f()` is called, then `this` is `obj` during the call of `f`. So it's either `user` or `admin` in the example above.

Calling without an object: `this == undefined`

We can even call the function without an object at all:

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined
```

In this case `this` is `undefined` in strict mode. If we try to access `this.name`, there will be an error.

In non-strict mode the value of `this` in such case will be the *global object* (`window` in a browser, we'll get to it later in the chapter [Global object](#)). This is a historical behavior that "use strict" fixes.

Usually such call is a programming error. If there's `this` inside a function, it expects to be called in an object context.

The consequences of unbound `this`

If you come from another programming language, then you are probably used to the idea of a "bound `this`", where methods defined in an object always have `this` referencing that object.

In JavaScript `this` is "free", its value is evaluated at call-time and does not depend on where the method was declared, but rather on what object is "before the dot".

The concept of run-time evaluated `this` has both pluses and minuses. On the one hand, a function can be reused for different objects. On the other hand, the greater flexibility creates more possibilities for mistakes.

Here our position is not to judge whether this language design decision is good or bad. We'll understand how to work with it, how to get benefits and avoid problems.

Internals: Reference Type

In-depth language feature

This section covers an advanced topic, to understand certain edge-cases better.

If you want to go on faster, it can be skipped or postponed.

An intricate method call can lose `this`, for instance:

```
let user = {
  name: "John",
  hi() { alert(this.name); },
  bye() { alert("Bye"); }
};

user.hi(); // John (the simple call works)

// now let's call user.hi or user.bye depending on the name
(user.name == "John" ? user.hi : user.bye)(); // Error!
```

On the last line there is a conditional operator that chooses either `user.hi` or `user.bye`. In this case the result is `user.hi`.

Then the method is immediately called with parentheses `()`. But it doesn't work correctly!

As you can see, the call results in an error, because the value of `"this"` inside the call becomes `undefined`.

This works (object dot method):

```
user.hi();
```

This doesn't (evaluated method):

```
(user.name == "John" ? user.hi : user.bye)(); // Error!
```

Why? If we want to understand why it happens, let's get under the hood of how `obj.method()` call works.

Looking closely, we may notice two operations in `obj.method()` statement:

1. First, the dot `.` retrieves the property `obj.method`.
2. Then parentheses `()` execute it.

So, how does the information about `this` get passed from the first part to the second one?

If we put these operations on separate lines, then `this` will be lost for sure:

```
let user = {
  name: "John",
```

```

    hi() { alert(this.name); }
  }

// split getting and calling the method in two lines
let hi = user.hi;
hi(); // Error, because this is undefined

```

Here `hi = user.hi` puts the function into the variable, and then on the last line it is completely standalone, and so there's no `this`.

To make `user.hi()` calls work, JavaScript uses a trick – the dot `.` returns not a function, but a value of the special [Reference Type](#).

The Reference Type is a “specification type”. We can’t explicitly use it, but it is used internally by the language.

The value of Reference Type is a three-value combination (`base`, `name`, `strict`), where:

- `base` is the object.
- `name` is the property name.
- `strict` is true if `use strict` is in effect.

The result of a property access `user.hi` is not a function, but a value of Reference Type. For `user.hi` in strict mode it is:

```

// Reference Type value
(user, "hi", true)

```

When parentheses `()` are called on the Reference Type, they receive the full information about the object and its method, and can set the right `this` (`=user` in this case).

Reference type is a special “intermediary” internal type, with the purpose to pass information from dot `.` to calling parentheses `()`.

Any other operation like assignment `hi = user.hi` discards the reference type as a whole, takes the value of `user.hi` (a function) and passes it on. So any further operation “loses” `this`.

So, as the result, the value of `this` is only passed the right way if the function is called directly using a dot `obj.method()` or square brackets `obj['method']()` syntax (they do the same here). Later in this tutorial, we will learn various ways to solve this problem such as [func.bind\(\)](#).

Arrow functions have no “this”

Arrow functions are special: they don’t have their “own” `this`. If we reference `this` from such a function, it’s taken from the outer “normal” function.

For instance, here `arrow()` uses `this` from the outer `user.sayHi()` method:

```

let user = {
  firstName: "Ilya",
  sayHi() {

```

```
let arrow = () => alert(this.firstName);
arrow();
}
};

user.sayHi(); // Ilya
```

That's a special feature of arrow functions, it's useful when we actually do not want to have a separate `this`, but rather to take it from the outer context. Later in the chapter [Arrow functions revisited](#) we'll go more deeply into arrow functions.

Summary

- Functions that are stored in object properties are called “methods”.
- Methods allow objects to “act” like `object.doSomething()`.
- Methods can reference the object as `this`.

The value of `this` is defined at run-time.

- When a function is declared, it may use `this`, but that `this` has no value until the function is called.
- A function can be copied between objects.
- When a function is called in the “method” syntax: `object.method()`, the value of `this` during the call is `object`.

Please note that arrow functions are special: they have no `this`. When `this` is accessed inside an arrow function, it is taken from outside.

Object to primitive conversion

What happens when objects are added `obj1 + obj2`, subtracted `obj1 - obj2` or printed using `alert(obj)`?

In that case, objects are auto-converted to primitives, and then the operation is carried out.

In the chapter [Type Conversions](#) we've seen the rules for numeric, string and boolean conversions of primitives. But we left a gap for objects. Now, as we know about methods and symbols it becomes possible to fill it.

1. All objects are `true` in a boolean context. There are only numeric and string conversions.
2. The numeric conversion happens when we subtract objects or apply mathematical functions. For instance, `Date` objects (to be covered in the chapter [Date and time](#)) can be subtracted, and the result of `date1 - date2` is the time difference between two dates.
3. As for the string conversion – it usually happens when we output an object like `alert(obj)` and in similar contexts.

ToPrimitive

We can fine-tune string and numeric conversion, using special object methods.

There are three variants of type conversion, so-called “hints”, described in the [specification](#) ↗ :

"string"

For an object-to-string conversion, when we're doing an operation on an object that expects a string, like `alert`:

```
// output
alert(obj);

// using object as a property key
anotherObj[obj] = 123;
```

"number"

For an object-to-number conversion, like when we're doing maths:

```
// explicit conversion
let num = Number(obj);

// maths (except binary plus)
let n = +obj; // unary plus
let delta = date1 - date2;

// less/greater comparison
let greater = user1 > user2;
```

"default"

Occurs in rare cases when the operator is "not sure" what type to expect.

For instance, binary plus `+` can work both with strings (concatenates them) and numbers (adds them), so both strings and numbers would do. So if the a binary plus gets an object as an argument, it uses the `"default"` hint to convert it.

Also, if an object is compared using `==` with a string, number or a symbol, it's also unclear which conversion should be done, so the `"default"` hint is used.

```
// binary plus uses the "default" hint
let total = obj1 + obj2;

// obj == number uses the "default" hint
if (user == 1) { ... };
```

The greater and less comparison operators, such as `<` `>`, can work with both strings and numbers too. Still, they use the `"number"` hint, not `"default"`. That's for historical reasons.

In practice though, we don't need to remember these peculiar details, because all built-in objects except for one case (`Date` object, we'll learn it later) implement `"default"` conversion the same way as `"number"`. And we can do the same.

No "boolean" hint

Please note – there are only three hints. It's that simple.

There is no "boolean" hint (all objects are `true` in boolean context) or anything else. And if we treat `"default"` and `"number"` the same, like most built-ins do, then there are only two conversions.

To do the conversion, JavaScript tries to find and call three object methods:

1. Call `obj[Symbol.toPrimitive](hint)` – the method with the symbolic key `Symbol.toPrimitive` (system symbol), if such method exists,
2. Otherwise if hint is `"string"`
 - try `obj.toString()` and `obj.valueOf()`, whatever exists.
3. Otherwise if hint is `"number"` or `"default"`
 - try `obj.valueOf()` and `obj.toString()`, whatever exists.

Symbol.toPrimitive

Let's start from the first method. There's a built-in symbol named `Symbol.toPrimitive` that should be used to name the conversion method, like this:

```
obj[Symbol.toPrimitive] = function(hint) {  
  // must return a primitive value  
  // hint = one of "string", "number", "default"  
};
```

For instance, here `user` object implements it:

```
let user = {  
  name: "John",  
  money: 1000,  
  
  [Symbol.toPrimitive](hint) {  
    alert(`hint: ${hint}`);  
    return hint == "string" ? `{name: "${this.name}"}` : this.money;  
  }  
};  
  
// conversions demo:  
alert(user); // hint: string -> {name: "John"}  
alert(+user); // hint: number -> 1000  
alert(user + 500); // hint: default -> 1500
```

As we can see from the code, `user` becomes a self-descriptive string or a money amount depending on the conversion. The single method `user[Symbol.toPrimitive]` handles all conversion cases.

toString/valueOf

Methods `toString` and `valueOf` come from ancient times. They are not symbols (symbols did not exist that long ago), but rather “regular” string-named methods. They provide an alternative “old-style” way to implement the conversion.

If there's no `Symbol.toPrimitive` then JavaScript tries to find them and try in the order:

- `toString` -> `valueOf` for “string” hint.
- `valueOf` -> `toString` otherwise.

These methods must return a primitive value. If `toString` or `valueOf` returns an object, then it's ignored (same as if there were no method).

By default, a plain object has following `toString` and `valueOf` methods:

- The `toString` method returns a string `"[object Object]"`.
- The `valueOf` method returns the object itself.

Here's the demo:

```
let user = {name: "John"};

alert(user); // [object Object]
alert(user.valueOf() === user); // true
```

So if we try to use an object as a string, like in an `alert` or so, then by default we see `[object Object]`.

And the default `valueOf` is mentioned here only for the sake of completeness, to avoid any confusion. As you can see, it returns the object itself, and so is ignored. Don't ask me why, that's for historical reasons. So we can assume it doesn't exist.

Let's implement these methods.

For instance, here `user` does the same as above using a combination of `toString` and `valueOf` instead of `Symbol.toPrimitive`:

```
let user = {
  name: "John",
  money: 1000,

  // for hint="string"
  toString() {
    return `${name: "${this.name}"}`;
  },

  // for hint="number" or "default"
  valueOf() {
    return this.money;
  }
};

alert(user); // toString -> {name: "John"}
alert(+user); // valueOf -> 1000
alert(user + 500); // valueOf -> 1500
```

As we can see, the behavior is the same as the previous example with `Symbol.toPrimitive`. Often we want a single “catch-all” place to handle all primitive conversions. In this case, we can implement `toString` only, like this:

```
let user = {
  name: "John",

  toString() {
    return this.name;
  }
};

alert(user); // toString -> John
alert(user + 500); // toString -> John500
```

In the absence of `Symbol.toPrimitive` and `valueOf`, `toString` will handle all primitive conversions.

Return types

The important thing to know about all primitive-conversion methods is that they do not necessarily return the “hinted” primitive.

There is no control whether `toString` returns exactly a string, or whether `Symbol.toPrimitive` method returns a number for a hint `"number"`.

The only mandatory thing: these methods must return a primitive, not an object.

Historical notes

For historical reasons, if `toString` or `valueOf` returns an object, there’s no error, but such value is ignored (like if the method didn’t exist). That’s because in ancient times there was no good “error” concept in JavaScript.

In contrast, `Symbol.toPrimitive` *must* return a primitive, otherwise there will be an error.

Further conversions

As we know already, many operators and functions perform type conversions, e.g. multiplication `*` converts operands to numbers.

If we pass an object as an argument, then there are two stages:

1. The object is converted to a primitive (using the rules described above).
2. If the resulting primitive isn’t of the right type, it’s converted.

For instance:

```
let obj = {
  // toString handles all conversions in the absence of other methods
```



```
toString() {  
  return "2";  
}  
};  
  
alert(obj * 2); // 4, object converted to primitive "2", then multiplication made it a number
```

1. The multiplication `obj * 2` first converts the object to primitive (that's a string `"2"`).
2. Then `"2" * 2` becomes `2 * 2` (the string is converted to number).

Binary plus will concatenate strings in the same situation, as it gladly accepts a string:

```
let obj = {  
  toString() {  
    return "2";  
  }  
};  
  
alert(obj + 2); // 22 ("2" + 2), conversion to primitive returned a string => concatenation
```

Summary

The object-to-primitive conversion is called automatically by many built-in functions and operators that expect a primitive as a value.

There are 3 types (hints) of it:

- `"string"` (for `alert` and other operations that need a string)
- `"number"` (for maths)
- `"default"` (few operators)

The specification describes explicitly which operator uses which hint. There are very few operators that “don’t know what to expect” and use the `"default"` hint. Usually for built-in objects `"default"` hint is handled the same way as `"number"`, so in practice the last two are often merged together.

The conversion algorithm is:

1. Call `obj[Symbol.toPrimitive](hint)` if the method exists,
2. Otherwise if hint is `"string"`
 - try `obj.toString()` and `obj.valueOf()`, whatever exists.
3. Otherwise if hint is `"number"` or `"default"`
 - try `obj.valueOf()` and `obj.toString()`, whatever exists.

In practice, it's often enough to implement only `obj.toString()` as a “catch-all” method for all conversions that return a “human-readable” representation of an object, for logging or debugging purposes.

Constructor, operator "new"

The regular `{...}` syntax allows to create one object. But often we need to create many similar objects, like multiple users or menu items and so on.

That can be done using constructor functions and the `"new"` operator.

Constructor function

Constructor functions technically are regular functions. There are two conventions though:

1. They are named with capital letter first.
2. They should be executed only with `"new"` operator.

For instance:

```
function User(name) {  
  this.name = name;  
  this.isAdmin = false;  
}  
  
let user = new User("Jack");  
  
alert(user.name); // Jack  
alert(user.isAdmin); // false
```

When a function is executed with `new`, it does the following steps:

1. A new empty object is created and assigned to `this`.
2. The function body executes. Usually it modifies `this`, adds new properties to it.
3. The value of `this` is returned.

In other words, `new User(...)` does something like:

```
function User(name) {  
  // this = {}; (implicitly)  
  
  // add properties to this  
  this.name = name;  
  this.isAdmin = false;  
  
  // return this; (implicitly)  
}
```

So `let user = new User("Jack")` gives the same result as:

```
let user = {  
  name: "Jack",  
  isAdmin: false  
};
```

Now if we want to create other users, we can call `new User("Ann")`, `new User("Alice")` and so on. Much shorter than using literals every time, and also easy to read.

That's the main purpose of constructors – to implement reusable object creation code.

Let's note once again – technically, any function can be used as a constructor. That is: any function can be run with `new`, and it will execute the algorithm above. The “capital letter first” is a common agreement, to make it clear that a function is to be run with `new`.

new function() { ... }

If we have many lines of code all about creation of a single complex object, we can wrap them in constructor function, like this:

```
let user = new function() {  
  this.name = "John";  
  this.isAdmin = false;  
  
  // ...other code for user creation  
  // maybe complex logic and statements  
  // local variables etc  
};
```

The constructor can't be called again, because it is not saved anywhere, just created and called. So this trick aims to encapsulate the code that constructs the single object, without future reuse.

Constructor mode test: `new.target`

Advanced stuff

The syntax from this section is rarely used, skip it unless you want to know everything.

Inside a function, we can check whether it was called with `new` or without it, using a special `new.target` property.

It is empty for regular calls and equals the function if called with `new`:

```
function User() {  
  alert(new.target);  
}  
  
// without "new":  
User(); // undefined  
  
// with "new":  
new User(); // function User { ... }
```

That can be used inside the function to know whether it was called with `new`, “in constructor mode”, or without it, “in regular mode”.

We can also make both `new` and regular calls to do the same, like this:

```
function User(name) {
  if (!new.target) { // if you run me without new
    return new User(name); // ...I will add new for you
  }

  this.name = name;
}

let john = User("John"); // redirects call to new User
alert(john.name); // John
```

This approach is sometimes used in libraries to make the syntax more flexible. So that people may call the function with or without `new`, and it still works.

Probably not a good thing to use everywhere though, because omitting `new` makes it a bit less obvious what's going on. With `new` we all know that the new object is being created.

Return from constructors

Usually, constructors do not have a `return` statement. Their task is to write all necessary stuff into `this`, and it automatically becomes the result.

But if there is a `return` statement, then the rule is simple:

- If `return` is called with an object, then the object is returned instead of `this`.
- If `return` is called with a primitive, it's ignored.

In other words, `return` with an object returns that object, in all other cases `this` is returned.

For instance, here `return` overrides `this` by returning an object:

```
function BigUser() {

  this.name = "John";

  return { name: "Godzilla" }; // <-- returns this object
}

alert( new BigUser().name ); // Godzilla, got that object
```

And here's an example with an empty `return` (or we could place a primitive after it, doesn't matter):

```
function SmallUser() {

  this.name = "John";

  return; // <-- returns this
}

alert( new SmallUser().name ); // John
```

Usually constructors don't have a `return` statement. Here we mention the special behavior with returning objects mainly for the sake of completeness.

Omitting parentheses

By the way, we can omit parentheses after `new`, if it has no arguments:

```
let user = new User; // <-- no parentheses
// same as
let user = new User();
```

Omitting parentheses here is not considered a “good style”, but the syntax is permitted by specification.

Methods in constructor

Using constructor functions to create objects gives a great deal of flexibility. The constructor function may have parameters that define how to construct the object, and what to put in it.

Of course, we can add to `this` not only properties, but methods as well.

For instance, `new User(name)` below creates an object with the given `name` and the method `sayHi`:

```
function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert( "My name is: " + this.name );
  };
}

let john = new User("John");

john.sayHi(); // My name is: John

/*
john = {
  name: "John",
  sayHi: function() { ... }
}
*/
```

To create complex objects, there's a more advanced syntax, [classes](#), that we'll cover later.

Summary

- Constructor functions or, briefly, constructors, are regular functions, but there's a common agreement to name them with capital letter first.
- Constructor functions should only be called using `new`. Such a call implies a creation of empty `this` at the start and returning the populated one at the end.

We can use constructor functions to make multiple similar objects.

JavaScript provides constructor functions for many built-in language objects: like `Date` for dates, `Set` for sets and others that we plan to study.

i **Objects, we'll be back!**

In this chapter we only cover the basics about objects and constructors. They are essential for learning more about data types and functions in the next chapters.

After we learn that, we return to objects and cover them in-depth in the chapters [Prototypes](#), [inheritance](#) and [Classes](#).