CHAPTER 1

Lesson 5: Objects: The Basics

TOPICS

LEARNING OUTCOMES

At the end of the lesson, you should be able to:

1. Understand the concept of objects and how to create them.
2. Explore methods and the use of `this` within objects.
3. Learn how to use constructors and the `new` operator.
4. Utilize optional chaining to safely access object properties.

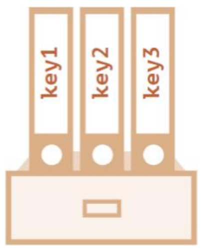
TOPIC 1.1: Objects

As we know from the chapter Data types, there are eight data types in JavaScript. Seven 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.

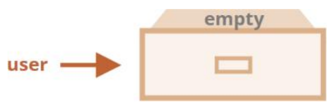
We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key. It’s easy to find a file by its name or add/remove a file.



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:

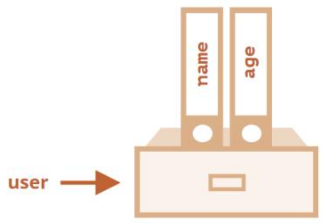
```
let user = {     // an object
  name: "John",  // by key "name" store value "John"
  age: 30        // by key "age" store value 30
};
```

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”.



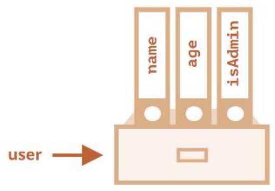
We can add, remove and read files from it at any time.

Property values are accessible using the dot notation:

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

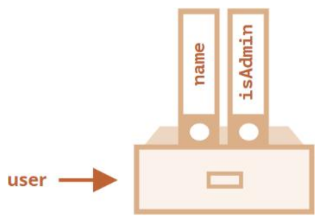
The value can be of any type. Let’s add a boolean one:

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



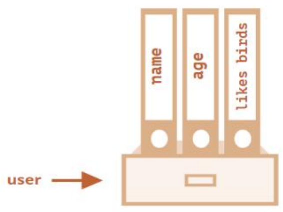
To remove a property, we can use the delete operator:

```
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 ($ and _ are allowed).

```
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 example:

```
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, when creating an object. That's called computed properties.

For example:

```
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;
```

Square brackets are much more powerful than 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 example:

```
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.

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

But for an object property, there’s no such restriction:

```
// these properties are all right
let obj = {
  for: 1,
  let: 2,
  return: 3
};

alert( obj.for + obj.let + obj.return );  // 6
```

In short, there are no limitations on property names. They can be any strings or symbols (a special type for identifiers, to be covered later).

Other types are automatically converted to strings.

For example, 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)
```

There’s a minor gotcha with a special property named __proto__. 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.

We’ll cover the special nature of __proto___in subsequent chapters, and suggest the ways to fix such behavior.

Property existence test, “in” operator.

A notable feature of objects in JavaScript, compared to many other languages, is that it’s possible to access any property. There will be no error if the property doesn’t exist!

Reading a non-existing property just returns undefined. So, we can easily test whether the property exists:

```
let user = {};

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

There's also a special operator "in" for that.

The syntax is:

```
"key" in object
```

For example:

```
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 means a variable should contain the actual name to be tested. For instance:

```
let user = { age: 30 };

let key = "age";
alert( key in user ); // true, property "age" exists
```

Why does the in operator exist? Isn't it enough to compare against undefined?

Well, most of the time the comparison with undefined works 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 should not be explicitly 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) {}

For example, let’s output all properties of user:

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

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.

TOPIC 1.2: 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 a function and assign it to the property user.sayHi of the object.

Then we can call it as user.sayHi(). The user can now speak!

A function that is a 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
```

!!! Object-Oriented Programming

When we write our code using objects to represent entites, that’s called Object-Oriented Programming or OOP.

OOP is a widely used programming style that most developers learn and practice. It is programming approach that uses classes and objects. It breaks down software into smaller, reusable pieces called classes, which can be used to create individual objects.

https://www.educative.io/blog/object-oriented-programming

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");
  }
};user.sayHi = sayHi;

user.sayHi(); // 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 example, 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 example.

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

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

};
```

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:

```
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(); // TypeError: Cannot read property 'name' of null
```

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, even if it’s not a method of an object.

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 example, 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.

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 example, 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.

TOPIC 1.3: Constructor, operator “new”

The regular {...} syntax allows us 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:

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

For example:

```
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 (except arrow functions, as they don’t have this) can be used as a constructor. It 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 an immediately called constructor function, like this:

```
// create a function and immediately call it with new
let user = new function() {
  this.name = "John";
  this.isAdmin = false;

  // ...other code for user creation
  // maybe complex logic and statements
  // local variables etc
};
```

This 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.

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

It is undefined 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 example, 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:

```
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 example, 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.

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.

TOPIC 1.4: Optional chaining '?.'

The optional chaining ?. is a safe way to access nested object properties, even if an intermediate property doesn't exist.

The "non-existing property" problem.

If you've just started to read the tutorial and learn JavaScript, maybe the problem hasn't touched you yet, but it's quite common.

As an example, let's say we have user objects that hold the information about our users.

Most of our users have addresses in user.address property, with the street user.address.street, but some did not provide them.

In such case, when we attempt to get user.address.street, and the user happens to be without an address, we get an error:

```
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() { ... }
}
*/
```

That’s the expected result. JavaScript works like this. As user.address is undefined, an attempt to get user.address.street fails with an error.

In many practical cases we’d prefer to get undefined instead of an error here (meaning “no street”).

and another example. In Web development, we can get an object that corresponds to a web page element using a special method call, such as document.querySelector('.elem'), and it returns null when there’s no such element.

```
// document.querySelector('.elem') is null if there's no element

let html = document.querySelector('.elem').innerHTML; // error if it's null
```

Once again, if the element doesn’t exist, we’ll get an error accessing .innerHTML property of null. And in some cases, when the absence of the element is normal, we’d like to avoid the error and just accept html = null as the result.

How can we do this?

The obvious solution would be to check the value using if or the conditional operator ?, before accessing its property, like this:

```
let user = {};

alert(user.address ? user.address.street : undefined);
```

It works, there’s no error… But it’s quite inelegant. As you can see, the "user.address" appears twice in the code.

Here’s how the same would look for document.querySelector:

```
let html = document.querySelector('.elem') ? document.querySelector('.elem').innerHTML : null;
```

We can see that the element search document.querySelector('.elem') is actually called twice here. Not good.

For more deeply nested properties, it becomes even uglier, as more repetitions are required.

E.g. let's get user.address.street.name in a similar fashion.

```
let user = {}; // user has no address

alert(user.address ? user.address.street ? user.address.street.name : null : null);
```

That's just awful, one may even have problems understanding such code.

There's a little better way to write it, using the && operator:

```
let user = {}; // user has no address

alert( user.address && user.address.street && user.address.street.name ); // undefined
(no error)
```

AND'ing the whole path to the property ensures that all components exist (if not, the evaluation stops), but also isn't ideal.

As you can see, property names are still duplicated in the code. E.g. in the code above, user.address appears three times.

That's why the optional chaining ?. was added to the language. To solve this problem once and for all!

Optional chaining.

The optional chaining ?. stops the evaluation if the value before ?. is undefined or null and returns undefined.

For brevity, we'll be saying that something "exists" if it's not null and not undefined.

In other words, value?.prop:

- works as value.prop, if value exists,
- otherwise (when value is undefined/null) it returns undefined.

Here's the safe way to access user.address.street using ?.:

```
let user = {}; // user has no address

alert( user?.address?.street ); // undefined (no error)
```

The code is short and clean, there's no duplication at all.

Here's an example with document.querySelector:asa

```
let html = document.querySelector('.elem')?.innerHTML;
```

Reading the address with user?.address works even if user object doesn’t exist:

```
let user = null;

alert( user?.address ); // undefined
alert( user?.address.street ); // undefined
```

Please note: the ?. syntax makes optional the value before it, but not any further.

E.g. in user?.address.street.name the ?. allows user to safely be null/undefined (and returns undefined in that case), but that’s only for user. Further properties are accessed in a regular way. If we want some of them to be optional, then we’ll need to replace more . with
?..

Don’t overuse the optional chaining.

We should use ?. only where it’s ok that something doesn’t exist.

For example, if according to our code logic user object must exist, but address is optional, then we should write user.address?.street, but not user?.address?.street.

Then, if user happens to be undefined, we’ll see a programming error about it and fix it. Otherwise, if we overuse ?., coding errors can be silenced where not appropriate, and become more difficult to debug.

The variable before ?. must be declared.

If there’s no variable user at all, then user?.anything triggers an error:

```
// ReferenceError: user is not defined

user?.address;
```

The variable must be declared (e.g. let/const/var user or as a function parameter). The optional chaining works only for declared variables.

Short-circuiting.

As it was said before, the ?. immediately stops (“short-circuits”) the evaluation if the left part doesn’t exist.

So, if there are any further function calls or operations to the right of ?., they won’t be made.

For example:

```
let user = null;
let x = 0;

user?.sayHi(x++); // no "user", so the execution doesn't reach sayHi call and x++

alert(x); // 0, value not incremented
```

Other variants: ?.(), ?.[]

The optional chaining ?. is not an operator, but a special syntax construct, that also works with functions and square brackets.

For example, ?.() is used to call a function that may not exist.

In the code below, some of our users have admin method, and some don’t:

```
let userAdmin =
  { admin() {
    alert("I am admin");
  }
};

let userGuest = {};

userAdmin.admin?.(); // I am admin

userGuest.admin?.(); // nothing happens (no such method)
```

Here, in both lines we first use the dot (userAdmin.admin) to get admin property, because we assume that the user object exists, so it’s safe read from it.

Then ?.() checks the left part: if the admin function exists, then it runs (that’s so for userAdmin). Otherwise (for userGuest) the evaluation stops without errors.

The ?.[] syntax also works, if we’d like to use brackets [] to access properties instead of dot .. Similar to previous cases, it allows to safely read a property from an object that may not exist.

```
let key = "firstName";

let user1 =
  { firstName:
  "John"
};

let user2 = null;

alert( user1?.[key] ); // John
alert( user2?.[key] ); // undefined
```

Also we can use ?. with delete:

```
delete user?.name; // delete user.name if user exists
```

We can use ?. for safe reading and deleting, but not writing

The optional chaining ?. has no use on the left side of an assignment.

For example:

```
let user = null;

user?.name = "John"; // Error, doesn't work
// because it evaluates to: undefined = "John"
```

Summary

The optional chaining ?. syntax has three forms:

1. obj?.prop – returns obj.prop if obj exists, otherwise undefined.
2. obj?.[prop] – returns obj[prop] if obj exists, otherwise undefined.
3. obj.method?.() – calls obj.method() if obj.method exists, otherwise returns undefined.

As we can see, all of them are straightforward and simple to use. The ?. checks the left part for null/undefined and allows the evaluation to proceed if it's not so.

A chain of ?. allows to safely access nested properties.

Still, we should apply ?. carefully, only where it's acceptable, according to our code logic, that the left part doesn't exist. So that it won't hide programming errors from us, if they occur.

TOPIC 1.4: Symbol type

By specification, only two primitive types may serve as object property keys:

- string type, or
- symbol type.

Otherwise, if one uses another type, such as number, it's autoconverted to string. So that obj[1] is the same as obj["1"], and obj[true] is the same as obj["true"].

Until now we've been using only strings.

Now let's explore symbols, see what they can do for us.

Symbols.

A "symbol" represents a unique identifier.

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

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

Upon creation, we can give symbols 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 exactly the same description, they are different values. The description is just a label that doesn't affect anything.

For example, 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.

So, to summarize, a symbol is a "primitive unique value" with an optional description. Let's see where we can use them.

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 example, 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 example, 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 belong to another codebase, it’s unsafe to add fields to them, since we might affect pre-defined behavior in that other codebase. However, symbols cannot be accessed accidentally. The third-party code won’t be aware of newly defined symbols, so it’s safe to add symbols to the user objects.

Also, imagine that another script wants to have its own identifier inside user, for its own purposes.

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 an object literal

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

```
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 example:

```
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] ); // Direct: 123
```

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 example:

```
// 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.

Symbol.keyFor

We have seen that for global symbols, Symbol.for(key) returns a symbol by name. To do the opposite – return a name by global symbol – we can use: Symbol.keyFor(sym):

For example:

```
// 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 returns undefined.

That said, all symbols have the description property.

For example:

```
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. But most libraries, built-in functions and syntax constructs don't use these methods.