2 - Objects

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.

Two ways of declaring objects:

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

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

Property values are accessible using the dot notation:

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

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
};
```

Accessing multiword properties:

```
let user = {};

// set
user["likes birds"] = true;

// get
```

```
alert(user["likes birds"]); // true

// delete
delete user["likes birds"];
```

Computed properties

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

```
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"
```

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

Object instance declaration example:

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

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

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

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

Properties existence test

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"
```

Operator in

It is used to check if a property is included in an object:

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

alert( "age" in user ); // true, user.age exists
alert( "blabla" in user ); // false, user.blabla doesn't exist
```

Iterating an Object

```
for (key in object){
 // do something
}
```

Objects references and copying

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

let admin = user; // copy the reference
```

Doing something like this will make admin and user have the same reference (the object is not copied but the reference is)

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

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

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

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

Const objects can be modified

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

user.name = "Pete"; // (*)

alert(user.name); // Pete
```

In this case, the reference to user must never change, but the properties inside can.

How can we duplicate an object?

We can create a new object and replicate the structure of the existing one, by iterating over its properties and copying them on the primitive level.

```
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 object with the same content
clone.name = "Pete"; // changed the data in it

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

This can also be done using Object.assign()

- The first argument **dest** is a target object.
- Further arguments **src1, ..., srcN** (can be as many as needed) are source objects.
- It copies the properties of all source objects **src1, ..., srcN** into the target **dest**. In other words, properties of all arguments starting from the second are copied into the first object.
- The call returns dest.

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

We can also 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 }
```

This line copies all properties of user in an empty object and returns it:

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

Until now we assumed that all properties of user are primitive. But properties can be references to other objects.

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

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

StructuredClone(object) clones the object with all nested properties.

Here's how we can use it in our example:

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

let clone = structuredClone(user);

alert( user.sizes === clone.sizes ); // false, different objects

// user and clone are totally unrelated now
user.sizes.width = 60;    // change a property from one place
alert(clone.sizes.width); // 50, not related
```

structuredClone() also supports circular cloning, when soemthing like this happens:

```
let user = {};
// let's create a circular reference:
// user.me references the user itself
user.me = user;

let clone = structuredClone(user);
alert(clone.me === clone); // true
```

Assign methods to objects

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

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

user.sayHi(); // Hello!
```

Shorthand syntax:

```
// these objects do the same

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

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. **This is a pointer to the current istance of the object.**

This pointer can also be used in occasions like:

```
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)
```

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
```

Summary:

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.

Garbage Collection

The garbage collector will free all the data that becomes **unreachable**, meaning that there is no pointer to that kind of data (like an object that after being defined is then assigned to null).

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

let admin = user;
```

In this case admin and user points to the same data, so doing `user = null` would not make the data inside the object unreachable, because admin is still pointing to that data.

More info about reachability are available here, it's a really important concept to have in mind:
https://javascript.info/garbage-collection

New Conctructor Operator

Using {} allows us to create one object, but often we need to create manu similar objects. This can be done using the new operator.
Constructor functions technically are regular functions. There are two conventions though:

- **They are named with capital letter first.**
- **They should be executed only with "new" operator.**

```
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:

- A new empty object is created and assigned to `this`.
- The function body executes. Usually it modifies `this`, adds new properties to it.
- The value of this is returned. So that i have a pointer to the object that just got created.

```
let user = new User("Jack")
//is the same thing as
let user = {
  name: "Jack",
  isAdmin: false
};
```

The main usage of constructors is to implement reusable object creation mode.
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.

Return form 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

//If i have an empy return:

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.

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 the parentheses is not considered good style
```

The constructor could add to `this` also methods:

```
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 we use **classes** instead.

Non-existing property

non-existing property problem

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:

```
let user = {}; // a user without "address" property

alert(user.address.street); // Error!
```

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").

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);
```

To avoid bad syntax we use **Optional Changing**:

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

Further in this article, 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)
```

Here's an example with `document.querySelector`:

```
let html = document.querySelector('.elem')?.innerHTML; // will be undefined,
if there's no element

// another example:

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

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.

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
```

Do not use ?. for writing

```
let user = null;

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

Summary

The optional chaining ?. syntax has three forms:

- obj?.prop – returns obj.prop if obj exists, otherwise undefined.
- obj?.[prop] – returns obj[prop] if obj exists, otherwise undefined.
- 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.

Symbol type

Only two primitives 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"]`.

Symbols

A symbol is a unique identifier, we can create one using Symbol(), passing a description to it that is useful for debugging purposes:

```
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.

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

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

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

Symbols does not autoconvert 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.

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

//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
```

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 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. If we accidentaly add properties names that were already used in the codebase, we would overwrite the data, using symbols prevents this.

Using symbols in object literal

If we want to use a symbol in an object we need to use square brackets:

```
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

```
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
```

Global Symbols

There exists a global symbol registry that we can use if we want same named symbols to be same entities. 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.

```
// 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.

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.

Object to primitive

JavaScript doesn't allow you to customize how operators work on objects. In case of such operations, objects are auto-converted to primitives, and then the operation is carried out over these primitives and results in a primitive value.

That's an important limitation: the result of obj1 + obj2 (or another math operation) can't be another object!

In this chapter we'll cover how an object converts to primitive and how to customize it.

We have two purposes:

- It will allow us to understand what's going on in case of coding mistakes, when such an operation happened accidentally.
- There are exceptions, where such operations are possible and look good. E.g. subtracting or comparing dates (Date objects). We'll come across them later.

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, usually objects implement it the same way as "number")
The specification describes explicitly which operator uses which hint.

The conversion algorithm is:

Call objSymbol.toPrimitive if the method exists,
Otherwise if hint is "string"
try calling obj.toString() or obj.valueOf(), whatever exists.
Otherwise if hint is "number" or "default"
try calling obj.valueOf() or obj.toString(), whatever exists.
All these methods must return a primitive to work (if defined).

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

For a better understanding of how Object operations work, please read:
http://www.adequatelygood.com/Object-to-Primitive-Conversions-in-JavaScript.html

From this performance test is that it’s always best to call your object’s type-conversion methods directly, rather than relying on the interpreter to do the complex series of method calls and comparisons needed to do it automatically.