1.To declare a constant (unchanging) variable, use const
```
const myBirthday = '18.04.1982';
```

```
const COLOR_ORANGE = "#FF7F00";
```
Benefits:

COLOR_ORANGE is much easier to remember than "#FF7F00".
It is much easier to mistype "#FF7F00" than COLOR_ORANGE.
When reading the code, COLOR_ORANGE is much more meaningful than #FF7F00.
When should we use capitals for a constant and when should we name it normally? Let’s make that clear.

Being a “constant” just means that a variable’s value never changes.
const ? is like let, but the value of the variable can’t be changed.

2.Data types:
There are 8 basic data types in JavaScript.
- Seven primitive data types:
- number for numbers of any kind: integer or floating-point, integers are limited by ±(253-1).
- bigint for integer numbers of arbitrary length.
- string for strings. A string may have zero or more characters, there’s no separate single-character type.
- boolean for true/false.
- null for unknown values ? a standalone type that has a single value null.
- undefined for unassigned values ? a standalone type that has a single value undefined.
- symbol for unique identifiers.
- And one non-primitive data type:
- object for more complex data structures.

3.string
```
let name = "John";

// embed a variable
alert( `Hello, ${name}!` ); // Hello, John!

// embed an expression
alert( `the result is ${1 + 2}` ); // the result is 3
```

4.The “null” value
The special null value does not belong to any of the types described above.

It forms a separate type of its own which contains only the null value:

```
let age = null;
```
In JavaScript, null is not a “reference to a non-existing object” or a “null pointer” like in some other languages.

It’s just a special value which represents “nothing”, “empty” or “value unknown”.

The code above states that age is unknown.
5.The “undefined” value
The special value undefined also stands apart. It makes a type of its own, just like null.

The meaning of undefined is “value is not assigned”.

If a variable is declared, but not assigned, then its value is undefined:

```
let age;

alert(age); // shows "undefined"
```
6. We covered 3 browser-specific functions to interact with visitors:

alert
shows a message.
prompt
shows a message asking the user to input text. It returns the text or, if Cancel button or Esc is clicked, null.
confirm
```
let isBoss = confirm("Are you the boss?");

alert( isBoss ); // true if OK is pressed
```
shows a message and waits for the user to press “OK” or “Cancel”. It returns true for OK and false for Cancel/Esc.

```
7.alert( Number("   123   ") ); // 123
alert( Number("123z") );      // NaN (error reading a number at "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

8.Exponentiation **
The exponentiation operator a ** b raises a to the power of b

```
9. numbers
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"

let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4
alert( b ); // 4
alert( c ); // 4
```

10. Increment/decrement

```
let counter = 2;
counter++;        // works the same as counter = counter + 1, but is shorter
alert( counter ); // 3
```

Decrement -- decreases a variable by 1:

```
let counter = 2;
counter--;        // works the same as counter = counter - 1, but is shorter
alert( counter ); // 1
let counter = 1;
let a = ++counter; // (*)

alert(a); // 2
```

In the line (*), the prefix form ++counter increments counter and returns the new value, 2. So, the alert shows 2.

Now, let’s use the postfix form:

```
let counter = 1;
let a = counter++; // (*) changed ++counter to counter++

alert(a); // 1
```

In the line (*), the postfix form counter++ also increments counter but returns the old value (prior to increment). So, the alert shows 1

```
let counter = 0;
counter++;
++counter;
alert( counter ); // 2, the lines above did the same
```

If we’d like to increase a value and immediately use the result of the operator, we need the prefix form:

```
let counter = 0;
alert( ++counter ); // 1
```

If we’d like to increment a value but use its previous value, we need the postfix form:

```
let counter = 0;
alert( counter++ ); // 0

let a = 1, b = 1;

alert( ++a ); // 2, prefix form returns the new value
alert( b++ ); // 1, postfix form returns the old value

alert( a ); // 2, incremented once
alert( b ); // 2, incremented once
```

11.String comparison

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
```

For instance, case matters. A capital letter "A" is not equal to the lowercase "a". Which one is greater? The lowercase "a". Why? Because t

12.Comparison with null and undefined

For a strict equality check ===

These values are different, because each of them is a different type.

```
alert( null === undefined ); // false
```

For a non-strict check ==

There’s a special rule. These two are a “sweet couple”: they equal each other (in the sense of ==), but not any other value.

```
alert( null == undefined ); // true
```

For maths and other comparisons < > <= >=

null/undefined are converted to numbers: null becomes 0, while undefined becomes NaN.

Strange result: null vs 0

Let’s compare null with a zero:

```
alert( null > 0 );  // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) true
```

Mathematically, that’s strange. The last result states that "null is greater than or equal to zero", so in one of the comparisons above it

The reason is that an equality check == and comparisons > < >= <= work differently. Comparisons convert null to a number, treating it as 0.

On the other hand, the equality check == for undefined and null is defined such that, without any conversions, they equal each other and do

```
5 > 4 → true
"apple" > "pineapple" → false
"2" > "12" → true
undefined == null → true
undefined === null → false
null == "\n0\n" → false
null === +"\n0\n" → false
```

Some of the reasons:

```
Obviously, true.
Dictionary comparison, hence false. "a" is smaller than "p".
Again, dictionary comparison, first char "2" is greater than the first char "1".
Values null and undefined equal each other only.
Strict equality is strict. Different types from both sides lead to false.
Similar to (4), null only equals undefined.
Strict equality of different types.
```

12. The “if” statement
if statement and the conditional operator ?, that’s also called a “question mark” operator.
if (year == 2015) alert('You are right!');
The if (…) statement evaluates the expression in its parentheses and converts the result to a boolean.

A number 0, an empty string "", null, undefined, and NaN all become false. Because of that they are called “falsy” values.
Other values become true, so they are called “truthy”.

So, the code under this condition would never execute:

```
if (0) { // 0 is falsy
  ...
}
```

…and inside this condition ? it always will:

```
if (1) { // 1 is truthy
  ...
}
```

question mark” operator lets us do that in a shorter and simpler way.

The operator is represented by a question mark ?. Sometimes it’s called “ternary”, because the operator has three operands. It is actually

The syntax is:

```
let result = condition ? value1 : value2;
```

The condition is evaluated: if it’s truthy then value1 is returned, otherwise ? value2.

```
let accessAllowed = age > 18 ? true : false;
let accessAllowed = (age > 18) ? true : false;
```

Please note:
In the example above, you can avoid using the question mark operator because the comparison itself returns true/false:

```
// the same
let accessAllowed = age > 18;
```

Multiple ‘?’
A sequence of question mark operators ? can return a value that depends on more than one condition.

For instance:

```
let age = prompt('age?', 18);

let message = (age < 3) ? 'Hi, baby!' :
  (age < 18) ? 'Hello!' :
  (age < 100) ? 'Greetings!' :
  'What an unusual age!';

alert( message );
```

13.logical operator

```
alert( 1 || 0 ); // 1 (1 is truthy)

alert( null || 1 ); // 1 (1 is the first truthy value)
alert( null || 0 || 1 ); // 1 (the first truthy value)

alert( undefined || null || 0 ); // 0 (all falsy, returns the last value)
```

In classical programming, AND returns true if both operands are truthy and false otherwise:

```
alert( true && true );   // true
alert( false && true );  // false
alert( true && false );  // false
alert( false && false ); // false

if (1 && 0) { // evaluated as true && false
  alert( "won't work, because the result is falsy" );
}

// if the first operand is truthy,
// AND returns the second operand:
alert( 1 && 0 ); // 0
alert( 1 && 5 ); // 5

// if the first operand is falsy,
// AND returns it. The second operand is ignored
alert( null && 5 ); // null
alert( 0 && "no matter what" ); // 0
```

We can also pass several values in a row. See how the first falsy one is returned:

```
alert( 1 && 2 && null && 3 ); // null
```

When all values are truthy, the last value is returned:

```
alert( 1 && 2 && 3 ); // 3, the last one
```

*So the code a && b || c && d is essentially the same as if the && expressions were in parentheses: (a && b) || (c && d)
! (NOT)
The boolean NOT operator is represented with an exclamation sign !.

The syntax is pretty simple:

result = !value;
The operator accepts a single argument and does the following:

Converts the operand to boolean type: true/false.
Returns the inverse value.
For instance:

```
alert( !true ); // false
alert( !0 ); // true
```

The answer is 2, that’s the first truthy value.

```
alert( null || 2 || undefined );
```

The answer: null, because it’s the first falsy value from the list.

```
alert(1 && null && 2);
```

```
alert( null || 2 && 3 || 4 );
```
The precedence of AND && is higher than ||, so it executes first.

The result of 2 && 3 = 3, so the expression becomes:

```
null || 3 || 4
```
Now the result is the first truthy value: 3.

14.For instance, a shorter way to write while (i != 0) is while (i):

```
let i = 3;
while (i) { // when i becomes 0, the condition becomes falsy, and the loop stops
  alert( i );
  i--;
}
```

A single execution of the loop body is called an iteration. The loop in the example above makes three iterations.

Curly braces are not required for a single-line body
If the loop body has a single statement, we can omit the curly braces {…}:

```
let i = 3;
while (i) alert(i--);
```

The general loop algorithm works like this:

```
Run begin
→ (if condition → run body and run step)
→ (if condition → run body and run step)
→ (if condition → run body and run step)
→ ...
```
That is, begin executes once, and then it iterates: after each condition test, body and step are executed.

We can also remove the step part:

```
let i = 0;

for (; i < 3;) {
  alert( i++ );
}
```
This makes the loop identical to while (i < 3).

We can actually remove everything, creating an infinite loop:

```
for (;;) {
  // repeats without limits
}
```

No break/continue to the right side of ‘?’
Please note that syntax constructs that are not expressions cannot be used with the ternary operator ?. In particular, directives such as b

For example, if we take this code:

```
if (i > 5) {
  alert(i);
} else {
  continue;
}
```
…and rewrite it using a question mark:

```
(i > 5) ? alert(i) : continue; // continue isn't allowed here
```

15.if we want the same code to run for case 3 and case 5:

```
let a = 3;

switch (a) {
  case 4:
    alert('Right!');
    break;

  case 3: // (*) grouped two cases
```

```
  case 5:
    alert('Wrong!');
    alert("Why don't you take a math class?");
    break;

  default:
    alert('The result is strange. Really.');
}
```

Now both 3 and 5 show the same message.

The ability to “group” cases is a side effect of how switch/case works without break. Here the execution of case 3 starts from the line (*)

16.Functions

Quite often we need to perform a similar action in many places of the script.

For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else.

Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.

We’ve already seen examples of built-in functions, like alert(message), prompt(message, default) and confirm(question). But we can create f

If a function is called, but an argument is not provided, then the corresponding value becomes undefined.

```
function sayHi() {   // (1) create
  alert( "Hello" );
}

let func = sayHi;    // (2) copy

func(); // Hello     // (3) run the copy (it works)!
sayHi(); // Hello    //     this still works too (why wouldn't it)
```

Why is there a semicolon at the end?

You might wonder, why do Function Expressions have a semicolon ; at the end, but Function Declarations do not:

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

The answer is simple: a Function Expression is created here as function(…) {…} inside the assignment statement: let sayHi = …;. The semicol

The semicolon would be there for a simpler assignment, such as let sayHi = 5;, and it’s also there for a function assignment.

The function should ask the question and, depending on the user’s answer, call yes() or no():

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "You agreed." );
}

function showCancel() {
  alert( "You canceled the execution." );
}

// usage: functions showOk, showCancel are passed as arguments to ask
ask("Do you agree?", showOk, showCancel);
```

In practice, such functions are quite useful. The major difference between a real-life ask and the example above is that real-life function

The arguments showOk and showCancel of ask are called callback functions or just callbacks.

The idea is that we pass a function and expect it to be “called back” later if necessary. In our case, showOk becomes the callback for “yes

*A Function Declaration can be called earlier than it is defined.

For example, a global Function Declaration is visible in the whole script, no matter where it is.

For example, this works:

```
sayHi("John"); // Hello, John

function sayHi(name) {
  alert( `Hello, ${name}` );
}
```

The Function Declaration sayHi is created when JavaScript is preparing to start the script and is visible everywhere in it.

…If it were a Function Expression, then it wouldn’t work:

```
sayHi("John"); // error!

let sayHi = function(name) {  // (*) no magic any more
  alert( `Hello, ${name}` );
};
```

*Another special feature of Function Declarations is their block scope.

In strict mode, when a Function Declaration is within a code block, it’s visible everywhere inside that block. But not outside of it.

For instance, let’s imagine that we need to declare a function welcome() depending on the age variable that we get during runtime. And then

If we use Function Declaration, it won’t work as intended:

```
let age = prompt("What is your age?", 18);

// conditionally declare a function
if (age < 18) {

  function welcome() {
    alert("Hello!");
  }

} else {

  function welcome() {
    alert("Greetings!");
  }

}

// ...use it later
welcome(); // Error: welcome is not defined
```

That’s because a Function Declaration is only visible inside the code block in which it resides.

Here’s another example:

```
let age = 16; // take 16 as an example

if (age < 18) {
  welcome();               // \   (runs)
                           //  |
  function welcome() {     //  |
    alert("Hello!");       //  |  Function Declaration is available
  }                        //  |  everywhere in the block where it's declared
                           //  |
  welcome();               // /   (runs)

} else {

  function welcome() {
    alert("Greetings!");
  }
}

// Here we're out of curly braces,
// so we can not see Function Declarations made inside of them.

welcome(); // Error: welcome is not defined
```

What can we do to make welcome visible outside of if?

The correct approach would be to use a Function Expression and assign welcome to the variable that is declared outside of if and has the pr

This code works as intended:

```
let age = prompt("What is your age?", 18);

let welcome;

if (age < 18) {

  welcome = function() {
    alert("Hello!");
  };

} else {

  welcome = function() {
    alert("Greetings!");
  };

}

welcome(); // ok now
```

Or we could simplify it even further using a question mark operator ?:

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  function() { alert("Hello!"); } :
  function() { alert("Greetings!"); };

welcome(); // ok now
```

*Arrow functions, the basics

There’s another very simple and concise syntax for creating functions, that’s often better than Function Expressions.

```
let double = n => n * 2;
// roughly the same as: let double = function(n) { return n * 2 }
```

```
alert( double(3) ); // 6
If there are no arguments, parentheses are empty, but they must be present:

let sayHi = () => alert("Hello!");

sayHi();
Arrow functions can be used in the same way as Function Expressions.

For instance, to dynamically create a function:

let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  () => alert('Hello!') :
  () => alert("Greetings!");

welcome();

17.Semicolons are not required after code blocks {...} and syntax constructs with them like loops:

Variables
Can be declared using:

let
const (constant, can’t be changed)
var (old-style, will see later)

18.note (Code quality) next time will entry
19. Object

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

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
};

let user = {};

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

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

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

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

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

*"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

*for in loop
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
}
To access a property, we can use:

The dot notation: obj.property.
Square brackets notation obj["property"]. Square brackets allow taking 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.
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.

```
let salaries = {
  John: 100,
  Ann: 160,
  Pete: 130
};

let sum = 0;
for (let key in salaries) {
  sum += salaries[key];
}

alert(sum); // 390
```

20.Object references and copying

```
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
```

We also can use Object.assign to replace for..in 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.

Now it's not enough to copy clone.sizes = user.sizes, because user.sizes is an object, and will be copied by reference, so clone and user w

```
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, get the result from the other one
```

To make a "real copy" (a clone) we can use Object.assign

22. if you want to undrstand this object Garbage collection chapter you should entry the url
https://javascript.info/garbage-collection

23.Object methods
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
user.sayHi = sayHi;

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

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().

*"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

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(); // 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 a

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.

```
*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.

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.

22.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 item

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

Constructor function
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.
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:

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.
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

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

```
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

```
// 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 co

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 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, an

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

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 th

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 comp

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 h

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

****data types :

23.Methods of primitives

Is a value of a primitive type.

There are 7 primitive types: string, number, bigint, boolean, symbol, null and undefined.

In order for that to work, a special “object wrapper” that provides the extra functionality is created, and then is destroyed.

The “object wrappers” are different for each primitive type and are called: String, Number, Boolean, Symbol and BigInt. Thus, they provide

For instance, there exists a string method str.toUpperCase() that returns a capitalized str.

Here’s how it works:

```
let str = "Hello";

alert( str.toUpperCase() ); // HELLO
```

Simple, right? Here’s what actually happens in str.toUpperCase():

The string str is a primitive. So in the moment of accessing its property, a special object is created that knows the value of the string,

That method runs and returns a new string (shown by alert).

The special object is destroyed, leaving the primitive str alone.

null/undefined have no methods

The special primitives null and undefined are exceptions. They have no corresponding “wrapper objects” and provide no methods. In a sense,

An attempt to access a property of such value would give the error:

```
alert(null.test); // error
```

24. Numbers : if you to understand clearly you can check out here

https://javascript.info/number

25.Rounding

One of the most used operations when working with numbers is rounding.

There are several built-in functions for rounding:

Math.floor

Rounds down: 3.1 becomes 3, and -1.1 becomes -2.

Math.ceil

Rounds up: 3.1 becomes 4, and -1.1 becomes -1.

Math.round

Rounds to the nearest integer: 3.1 becomes 3, 3.6 becomes 4, the middle case: 3.5 rounds up to 4 too.

Math.trunc (not supported by Internet Explorer)

Removes anything after the decimal point without rounding: 3.1 becomes 3, -1.1 becomes -1.

There are two ways to do so:

Multiply-and-divide.

For example, to round the number to the 2nd digit after the decimal, we can multiply the number by 100, call the rounding function and then

```
let num = 1.23456;

alert( Math.round(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

The method toFixed(n) rounds the number to n digits after the point and returns a string representation of the result.

```
let num = 12.34;
alert( num.toFixed(1) ); // "12.3"
```

This rounds up or down to the nearest value, similar to Math.round:

```
let num = 12.36;
alert( num.toFixed(1) ); // "12.4"
```

Please note that the result of toFixed is a string. If the decimal part is shorter than required, zeroes are appended to the end:

```
let num = 12.34;
alert( num.toFixed(5) ); // "12.34000", added zeroes to make exactly 5 digits
```

We can convert it to a number using the unary plus or a Number() call, e.g write +num.toFixed(5).

parseInt and parseFloat

Numeric conversion using a plus + or Number() is strict. If a value is not exactly a number, it fails:

```
alert( +"100px" ); // NaN
```

The sole exception is spaces at the beginning or at the end of the string, as they are ignored.

But in real life we often have values in units, like "100px" or "12pt" in CSS. Also in many countries the currency symbol goes after the am

That’s what parseInt and parseFloat are for.

They “read” a number from a string until they can’t. In case of an error, the gathered number is returned. The function parseInt returns an

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5

alert( parseInt('12.3') ); // 12, only the integer part is returned
alert( parseFloat('12.3.4') ); // 12.3, the second point stops the reading
```

There are situations when parseInt/parseFloat will return NaN. It happens when no digits could be read:

```
alert( parseInt('a123') ); // NaN, the first symbol stops the process
```

The second argument of parseInt(str, radix)

The parseInt() function has an optional second parameter. It specifies the base of the numeral system, so parseInt can also parse strings o

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, without 0x also works

alert( parseInt('2n9c', 36) ); // 123456
```

25.String

length is a property

People with a background in some other languages sometimes mistype by calling str.length() instead of just str.length. That doesn’t work.

Please note that str.length is a numeric property, not a function. There is no need to add parenthesis after it.

Accessing characters

To get a character at position pos, use square brackets [pos] or call the method str.charAt(pos). The first character starts from the zero

```
let str = `Hello`;

// the first character
alert( str[0] ); // H
alert( str.charAt(0) ); // H

// the last character
alert( str[str.length - 1] ); // o
```

The square brackets are a modern way of getting a character, while charAt exists mostly for historical reasons.

The only difference between them is that if no character is found, [] returns undefined, and charAt returns an empty string:

```
let str = `Hello`;

alert( str[1000] ); // undefined
alert( str.charAt(1000) ); // '' (an empty string)
```

*Strings are immutable

Strings can’t be changed in JavaScript. It is impossible to change a character.

Let’s try it to show that it doesn’t work:

```
let str = 'Hi';

str[0] = 'h'; // error
alert( str[0] ); // doesn't work
```

The usual workaround is to create a whole new string and assign it to str instead of the old one.

For instance:

```
let str = 'Hi';

str = 'h' + str[1]; // replace the string

alert( str ); // hi
```

*Changing the case

Methods toLowerCase() and toUpperCase() change the case:

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

Or, if we want a single character lowercased:

```
alert( 'Interface'[0].toLowerCase() ); // 'i'
```

*Searching for a substring

There are multiple ways to look for a substring within a string.

str.indexOf

The first method is str.indexOf(substr, pos).

It looks for the substr in str, starting from the given position pos, and returns the position where the match was found or -1 if nothing c

For instance:

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, because 'Widget' is found at the beginning
alert( str.indexOf('widget') ); // -1, not found, the search is case-sensitive

alert( str.indexOf("id") ); // 1, "id" is found at the position 1 (..idget with id)
```

The optional second parameter allows us to start searching from a given position.

For instance, the first occurrence of "id" is at position 1. To look for the next occurrence, let’s start the search from position 2:

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

If we’re interested in all occurrences, we can run indexOf in a loop. Every new call is made with the position after the previous match:

```
let str = 'As sly as a fox, as strong as an ox';

let target = 'as'; // let's look for it

let pos = 0;
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert( `Found at ${foundPos}` );
  pos = foundPos + 1; // continue the search from the next position
}
```

The same algorithm can be layed out shorter:

```
let str = "As sly as a fox, as strong as an ox";
let target = "as";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( pos );
}
```

str.lastIndexOf(substr, position)

There is also a similar method str.lastIndexOf(substr, position) that searches from the end of a string to its beginning.

It would list the occurrences in the reverse order.

There is a slight inconvenience with indexOf in the if test. We can’t put it in the if like this:

```
let str = "Widget with id";

if (str.indexOf("Widget")) {
    alert("We found it"); // doesn't work!
}
```

The alert in the example above doesn’t show because str.indexOf("Widget") returns 0 (meaning that it found the match at the starting positi

So, we should actually check for -1, like this:

```
let str = "Widget with id";

if (str.indexOf("Widget") != -1) {
    alert("We found it"); // works now!
}
```

*includes, startsWith, endsWith

The more modern method str.includes(substr, pos) returns true/false depending on whether str contains substr within.

It’s the right choice if we need to test for the match, but don’t need its position:

```
alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false
```

The optional second argument of str.includes is the position to start searching from:

```
alert( "Widget".includes("id") ); // true
alert( "Widget".includes("id", 3) ); // false, from position 3 there is no "id"
```

The methods str.startsWith and str.endsWith do exactly what they say:

```
alert( "Widget".startsWith("Wid") ); // true, "Widget" starts with "Wid"
alert( "Widget".endsWith("get") ); // true, "Widget" ends with "get"
```

*Getting a substring

There are 3 methods in JavaScript to get a substring: substring, substr and slice.

str.slice(start [, end])

Returns the part of the string from start to (but not including) end.

For instance:

```
let str = "stringify";
alert( str.slice(0, 5) ); // 'strin', the substring from 0 to 5 (not including 5)
alert( str.slice(0, 1) ); // 's', from 0 to 1, but not including 1, so only character at 0
```

If there is no second argument, then slice goes till the end of the string:

```
let str = "stringify";
alert( str.slice(2) ); // 'ringify', from the 2nd position till the end
```

Negative values for start/end are also possible. They mean the position is counted from the string end:

```
let str = "stringify";

// start at the 4th position from the right, end at the 1st from the right
alert( str.slice(-4, -1) ); // 'gif'
```

str.substring(start [, end])
Returns the part of the string between start and end.

This is almost the same as slice, but it allows start to be greater than end.

For instance:

```
let str = "stringify";

// these are same for substring
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// ...but not for slice:
alert( str.slice(2, 6) ); // "ring" (the same)
alert( str.slice(6, 2) ); // "" (an empty string)
```

Negative arguments are (unlike slice) not supported, they are treated as 0.

str.substr(start [, length])
Returns the part of the string from start, with the given length.

In contrast with the previous methods, this one allows us to specify the length instead of the ending position:

```
let str = "stringify";
alert( str.substr(2, 4) ); // 'ring', from the 2nd position get 4 characters
```

The first argument may be negative, to count from the end:

```
let str = "stringify";
alert( str.substr(-4, 2) ); // 'gi', from the 4th position get 2 characters
```

This method resides in the Annex B of the language specification. It means that only browser-hosted Javascript engines should support it, a

Let’s recap these methods to avoid any confusion:

method	selects…	negatives
slice(start, end)	from start to end (not including end)	allows negatives
substring(start, end)	between start and end	negative values mean 0
substr(start, length)	from start get length characters	allows negative start

Which one to choose?
All of them can do the job. Formally, substr has a minor drawback: it is described not in the core JavaScript specification, but in Annex B

Of the other two variants, slice is a little bit more flexible, it allows negative arguments and shorter to write. So, it’s enough to remem

25.Array
Methods pop/push, shift/unshift
push appends an element to the end.
shift get an element from the beginning, advancing the queue, so that the 2nd element becomes the 1st.

push adds an element to the end.
pop takes an element from the end.

*Loops
One of the oldest ways to cycle array items is the for loop over indexes:

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

But for arrays there is another form of loop, for..of:

```
let fruits = ["Apple", "Orange", "Plum"];

// iterates over array elements
for (let fruit of fruits) {
  alert( fruit );
}
```

The for..of doesn’t give access to the number of the current element, just its value, but in most cases that’s enough. And it’s shorter.

Technically, because arrays are objects, it is also possible to use for..in:

```
let arr = ["Apple", "Orange", "Pear"];

for (let key in arr) {
  alert( arr[key] ); // Apple, Orange, Pear
}
```

But that’s actually a bad idea. There are potential problems with it:

The loop for..in iterates over all properties, not only the numeric ones.

There are so-called “array-like” objects in the browser and in other environments, that look like arrays. That is, they have length and ind

The for..in loop is optimized for generic objects, not arrays, and thus is 10-100 times slower. Of course, it’s still very fast. The speedu

Generally, we shouldn’t use for..in for arrays.

*A word about “length”
The length property automatically updates when we modify the array. To be precise, it is actually not the count of values in the array, but

For instance, a single element with a large index gives a big length:

```
let fruits = [];
fruits[123] = "Apple";

alert( fruits.length ); // 124
```

Note that we usually don’t use arrays like that.

Another interesting thing about the length property is that it’s writable.

If we increase it manually, nothing interesting happens. But if we decrease it, the array is truncated. The process is irreversible, here’s

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // truncate to 2 elements
alert( arr ); // [1, 2]

arr.length = 5; // return length back
alert( arr[3] ); // undefined: the values do not return
```

So, the simplest way to clear the array is: arr.length = 0;

*new Array()
There is one more syntax to create an array:

```
let arr = new Array("Apple", "Pear", "etc");
```

It’s rarely used, because square brackets [] are shorter. Also, there’s a tricky feature with it.

If new Array is called with a single argument which is a number, then it creates an array without items, but with the given length.

Let’s see how one can shoot themselves in the foot:

```
let arr = new Array(2); // will it create an array of [2] ?

alert( arr[0] ); // undefined! no elements.

alert( arr.length ); // length 2
```

To avoid such surprises, we usually use square brackets, unless we really know what we’re doing.

*toString
Arrays have their own implementation of toString method that returns a comma-separated list of elements.

For instance:

```
let arr = [1, 2, 3];

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

Also, let’s try this:

```
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

Arrays do not have Symbol.toPrimitive, neither a viable valueOf, they implement only toString conversion, so here [] becomes an empty strin

When the binary plus "+" operator adds something to a string, it converts it to a string as well, so the next step looks like this:

```
alert( "" + 1 ); // "1"
alert( "1" + 1 ); // "11"
alert( "1,2" + 1 ); // "1,21"
```

26.Don’t compare arrays with ==
if you want clearlt please check out this https://javascript.info/array#don-t-compare-arrays-with

27.for map set https://javascript.info/map-set

28.Destructuring assignment for details
https://javascript.info/destructuring-assignment
Array destructuring
Here’s an example of how an array is destructured into variables:

```
// we have an array with the name and surname
let arr = ["John", "Smith"]

// destructuring assignment
// sets firstName = arr[0]
// and surname = arr[1]
let [firstName, surname] = arr;

alert(firstName); // John
alert(surname);  // Smith
```

The rest ‘…’
Usually, if the array is longer than the list at the left, the “extra” items are omitted.

For example, here only two items are taken, and the rest is just ignored:

```
let [name1, name2] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius
alert(name2); // Caesar
// Further items aren't assigned anywhere
```

If we'd like also to gather all that follows – we can add one more parameter that gets "the rest" using three dots "...":

```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

// rest is array of items, starting from the 3rd one
alert(rest[0]); // Consul
alert(rest[1]); // of the Roman Republic
alert(rest.length); // 2
```

Object destructuring

The destructuring assignment also works with objects.

The basic syntax is:

```
let {var1, var2} = {var1:…, var2:…}
```

We should have an existing object on the right side, that we want to split into variables. The left side contains an object-like "pattern"

For instance:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

let {title, width, height} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

Properties options.title, options.width and options.height are assigned to the corresponding variables.

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

// { sourceProperty: targetVariable }
let {width: w, height: h, title} = options;

// width -> w
// height -> h
// title -> title

alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

The colon shows "what : goes where". In the example above the property width goes to w, property height goes to h, and title is assigned to

The rest pattern "…"

What if the object has more properties than we have variables? Can we take some and then assign the "rest" somewhere?

We can use the rest pattern, just like we did with arrays. It's not supported by some older browsers (IE, use Babel to polyfill it), but wo

It looks like this:

```
let options = {
  title: "Menu",
  height: 200,
  width: 100
};

// title = property named title
// rest = object with the rest of properties
let {title, ...rest} = options;

// now title="Menu", rest={height: 200, width: 100}
alert(rest.height);  // 200
alert(rest.width);   // 100

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

let {name, years: age, isAdmin = false} = user;

alert( name ); // John
alert( age ); // 30
alert( isAdmin ); // false
```

29.Date and time for details

https://javascript.info/date

Let's meet a new built-in object: Date. It stores the date, time and provides methods for date/time management.

For instance, we can use it to store creation/modification times, to measure time, or just to print out the current date.

Creation
To create a new Date object call new Date() with one of the following arguments:

new Date()
Without arguments – create a Date object for the current date and time:

```
let now = new Date();
alert( now ); // shows current date/time
```

Setting date components
The following methods allow to set date/time components:

setFullYear(year, [month], [date])
setMonth(month, [date])
setDate(date)
setHours(hour, [min], [sec], [ms])
setMinutes(min, [sec], [ms])
setSeconds(sec, [ms])
setMilliseconds(ms)
setTime(milliseconds) (sets the whole date by milliseconds since 01.01.1970 UTC)
Every one of them except setTime() has a UTC-variant, for instance: setUTCHours().

30.JSON.stringify for details
https://javascript.info/json
The JSON (JavaScript Object Notation) is a general format to represent values and objects. It is described as in RFC 4627 standard. Initial

JavaScript provides methods:

JSON.stringify to convert objects into JSON.
JSON.parse to convert JSON back into an object.
For instance, here we JSON.stringify a student:

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
  spouse: null
};

let json = JSON.stringify(student);

alert(typeof json); // we've got a string!

alert(json);
/* JSON-encoded object:
{
  "name": "John",
  "age": 30,
  "isAdmin": false,
  "courses": ["html", "css", "js"],
  "spouse": null
}
*/
```

The method JSON.stringify(student) takes the object and converts it into a string.

The resulting json string is called a JSON-encoded or serialized or stringified or marshalled object. We are ready to send it over the wire

Please note that a JSON-encoded object has several important differences from the object literal:

Strings use double quotes. No single quotes or backticks in JSON. So 'John' becomes "John".
Object property names are double-quoted also. That’s obligatory. So age:30 becomes "age":30.
JSON.stringify can be applied to primitives as well.

JSON supports following data types:

Objects { ... }
Arrays [...]
Primitives:
strings,
numbers,
boolean values true/false,
null.
For instance:

```
// a number in JSON is just a number
alert( JSON.stringify(1) ) // 1

// a string in JSON is still a string, but double-quoted
alert( JSON.stringify('test') ) // "test"

alert( JSON.stringify(true) ); // true

alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

JSON is data-only language-independent specification, so some JavaScript-specific object properties are skipped by JSON.stringify.

Namely:

Function properties (methods).
Symbolic keys and values.
Properties that store undefined.

JSON.parse
To decode a JSON-string, we need another method named JSON.parse.

The syntax:

```
let value = JSON.parse(str, [reviver]);
```

str
JSON-string to parse.
reviver
Optional function(key,value) that will be called for each (key, value) pair and can transform the value.
For instance:

```
// stringified array
let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1
```

Summary
JSON is a data format that has its own independent standard and libraries for most programming languages.
JSON supports plain objects, arrays, strings, numbers, booleans, and null.
JavaScript provides methods JSON.stringify to serialize into JSON and JSON.parse to read from JSON.
Both methods support transformer functions for smart reading/writing.
If an object has toJSON, then it is called by JSON.stringify.

31.Recursion and stack for details
https://javascript.info/recursion

32.Rest parameters and spread syntax
Many JavaScript built-in functions support an arbitrary number of arguments.

For instance:

Math.max(arg1, arg2, ..., argN) – returns the greatest of the arguments.
Object.assign(dest, src1, ..., srcN) – copies properties from src1..N into dest.
…and so on.
In this chapter we'll learn how to do the same. And also, how to pass arrays to such functions as parameters.

Rest parameters ...
A function can be called with any number of arguments, no matter how it is defined.

Like here:

```
function sum(a, b) {
  return a + b;
}

alert( sum(1, 2, 3, 4, 5) );
```

There will be no error because of "excessive" arguments. But of course in the result only the first two will be counted.

The rest of the parameters can be included in the function definition by using three dots ... followed by the name of the array that will c

For instance, to gather all arguments into array args:

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

We can choose to get the first parameters as variables, and gather only the rest.

Here the first two arguments go into variables and the rest go into titles array:

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julius Caesar

  // the rest go into titles array
  // i.e. titles = ["Consul", "Imperator"]
  alert( titles[0] ); // Consul
  alert( titles[1] ); // Imperator
  alert( titles.length ); // 2
}

showName("Julius", "Caesar", "Consul", "Imperator");
```

The rest parameters must be at the end
The rest parameters gather all remaining arguments, so the following does not make sense and causes an error:

```
function f(arg1, ...rest, arg2) { // arg2 after ...rest ?!
  // error
}
```

The ...rest must always be last.

The "arguments" variable
There is also a special array-like object named arguments that contains all arguments by their index.

For instance:

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );

  // it's iterable
  // for(let arg of arguments) alert(arg);
}

// shows: 2, Julius, Caesar
showName("Julius", "Caesar");

// shows: 1, Ilya, undefined (no second argument)
showName("Ilya");
```

In old times, rest parameters did not exist in the language, and using arguments was the only way to get all arguments of the function. And

But the downside is that although arguments is both array-like and iterable, it’s not an array. It does not support array methods, so we ca

Also, it always contains all arguments. We can’t capture them partially, like we did with rest parameters.

So when we need these features, then rest parameters are preferred.

Arrow functions do not have "arguments"
If we access the arguments object from an arrow function, it takes them from the outer “normal” function.

Here’s an example:

```
function f() {
  let showArg = () => alert(arguments[0]);
  showArg();
}

f(1); // 1
```

As we remember, arrow functions don’t have their own this. Now we know they don’t have the special arguments object either.

Spread syntax
We’ve just seen how to get an array from the list of parameters.

But sometimes we need to do exactly the reverse.

For instance, there’s a built-in function Math.max that returns the greatest number from a list:

```
alert( Math.max(3, 5, 1) ); // 5
```

Now let’s say we have an array [3, 5, 1]. How do we call Math.max with it?

Passing it “as is” won’t work, because Math.max expects a list of numeric arguments, not a single array:

```
let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN
```

And surely we can’t manually list items in the code Math.max(arr[0], arr[1], arr[2]), because we may be unsure how many there are. As our s

Spread syntax to the rescue! It looks similar to rest parameters, also using ..., but does quite the opposite.

When ...arr is used in the function call, it “expands” an iterable object arr into the list of arguments.

For Math.max:

```
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread turns array into a list of arguments)
```

We also can pass multiple iterables this way:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr1, ...arr2) ); // 8
```

We can even combine the spread syntax with normal values:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25
```

Also, the spread syntax can be used to merge arrays:

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, then arr, then 2, then arr2)
```

In the examples above we used an array to demonstrate the spread syntax, but any iterable will do.

For instance, here we use the spread syntax to turn the string into array of characters:

```
let str = "Hello";

alert( [...str] ); // H,e,l,l,o
```

The spread syntax internally uses iterators to gather elements, the same way as for..of does.

So, for a string, for..of returns characters and ...str becomes "H","e","l","l","o". The list of characters is passed to array initializer

For this particular task we could also use Array.from, because it converts an iterable (like a string) into an array:

```
let str = "Hello";

// Array.from converts an iterable into an array
alert( Array.from(str) ); // H,e,l,l,o
```

The result is the same as [...str].

But there’s a subtle difference between Array.from(obj) and [...obj]:

Array.from operates on both array-likes and iterables.
The spread syntax works only with iterables.
So, for the task of turning something into an array, Array.from tends to be more universal.

Copy an array/object
Remember when we talked about Object.assign() in the past?

It is possible to do the same thing with the spread syntax.

```
let arr = [1, 2, 3];

let arrCopy = [...arr]; // spread the array into a list of parameters
                        // then put the result into a new array

// do the arrays have the same contents?
alert(JSON.stringify(arr) === JSON.stringify(arrCopy)); // true

// are the arrays equal?
alert(arr === arrCopy); // false (not same reference)

// modifying our initial array does not modify the copy:
arr.push(4);
alert(arr); // 1, 2, 3, 4
alert(arrCopy); // 1, 2, 3
```

Note that it is possible to do the same thing to make a copy of an object:

```
let obj = { a: 1, b: 2, c: 3 };

let objCopy = { ...obj }; // spread the object into a list of parameters
                          // then return the result in a new object

// do the objects have the same contents?
alert(JSON.stringify(obj) === JSON.stringify(objCopy)); // true

// are the objects equal?
alert(obj === objCopy); // false (not same reference)

// modifying our initial object does not modify the copy:
obj.d = 4;
alert(JSON.stringify(obj)); // {"a":1,"b":2,"c":3,"d":4}
alert(JSON.stringify(objCopy)); // {"a":1,"b":2,"c":3}
```

This way of copying an object is much shorter than let objCopy = Object.assign({}, obj) or for an array let arrCopy = Object.assign([], arr

Summary
When we see "..." in the code, it is either rest parameters or the spread syntax.

There’s an easy way to distinguish between them:

When ... is at the end of function parameters, it’s “rest parameters” and gathers the rest of the list of arguments into an array.
When ... occurs in a function call or alike, it’s called a “spread syntax” and expands an array into a list.
Use patterns:

Rest parameters are used to create functions that accept any number of arguments.
The spread syntax is used to pass an array to functions that normally require a list of many arguments.
Together they help to travel between a list and an array of parameters with ease.

All arguments of a function call are also available in “old-style” arguments: array-like iterable object.

32.Variable scope, closure for details :
https://javascript.info/closure

33.Global object for details :
https://javascript.info/global-object

34.Function object, NFE for details :
https://javascript.info/function-object
As we already know, a function in JavaScript is a value.

Every value in JavaScript has a type. What type is a function?

In JavaScript, functions are objects.

A good way to imagine functions is as callable “action objects”. We can not only call them, but also treat them as objects: add/remove prop

35.The "new Function" syntax
There’s one more way to create a function. It’s rarely used, but sometimes there’s no alternative.

Syntax
The syntax for creating a function:

let func = new Function ([arg1, arg2, ...argN], functionBody);
The function is created with the arguments arg1...argN and the given functionBody.

It’s easier to understand by looking at an example. Here’s a function with two arguments:

```
let sum = new Function('a', 'b', 'return a + b');

alert( sum(1, 2) ); // 3
```

And here there’s a function without arguments, with only the function body:

```
let sayHi = new Function('alert("Hello")');

sayHi(); // Hello
```

The major difference from other ways we’ve seen is that the function is created literally from a string, that is passed at run time.

All previous declarations required us, programmers, to write the function code in the script.

But new Function allows to turn any string into a function. For example, we can receive a new function from a server and then execute it:

```
let str = ... receive the code from a server dynamically ...

let func = new Function(str);
func();
```

It is used in very specific cases, like when we receive code from a server, or to dynamically compile a function from a template, in comple

36.Scheduling: setTimeout and setInterval
We may decide to execute a function not right now, but at a certain time later. That’s called “scheduling a call”.

There are two methods for it:

setTimeout allows us to run a function once after the interval of time.
setInterval allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval.
These methods are not a part of JavaScript specification. But most environments have the internal scheduler and provide these methods. In p

setTimeout
The syntax:

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

Parameters:

func|code
Function or a string of code to execute. Usually, that’s a function. For historical reasons, a string of code can be passed, but that’s not
delay
The delay before run, in milliseconds (1000 ms = 1 second), by default 0.
arg1, arg2…
Arguments for the function (not supported in IE9-)
For instance, this code calls sayHi() after one second:

```
function sayHi() {
  alert('Hello');
}

setTimeout(sayHi, 1000);
```

With arguments:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

If the first argument is a string, then JavaScript creates a function from it.

So, this will also work:

```
setTimeout("alert('Hello')", 1000);
```

But using strings is not recommended, use arrow functions instead of them, like this:

```
setTimeout(() => alert('Hello'), 1000);
```

Pass a function, but don’t run it
Novice developers sometimes make a mistake by adding brackets () after the function:

```
// wrong!
setTimeout(sayHi(), 1000);
```

That doesn’t work, because setTimeout expects a reference to a function. And here sayHi() runs the function, and the result of its executio

37.Canceling with clearTimeout
A call to setTimeout returns a “timer identifier” timerId that we can use to cancel the execution.

The syntax to cancel:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

In the code below, we schedule the function and then cancel it (changed our mind). As a result, nothing happens:

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier

clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```

As we can see from alert output, in a browser the timer identifier is a number. In other environments, this can be something else. For inst

Again, there is no universal specification for these methods, so that’s fine.

For browsers, timers are described in the timers section of HTML5 standard.

setInterval
The setInterval method has the same syntax as setTimeout:

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

All arguments have the same meaning. But unlike setTimeout it runs the function not only once, but regularly after the given interval of ti

To stop further calls, we should call clearInterval(timerId).

The following example will show the message every 2 seconds. After 5 seconds, the output is stopped:

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Time goes on while alert is shown
In most browsers, including Chrome and Firefox the internal timer continues “ticking” while showing alert/confirm/prompt.

So if you run the code above and don’t dismiss the alert window for some time, then the next alert will be shown immediately as you do it.

Nested setTimeout for details :
https://javascript.info/settimeout-setinterval

38.Decorators and forwarding, call/apply for details :
https://javascript.info/call-apply-decorators

39. The JavaScript languageAdvanced working with functions
October 21, 2021
Function binding
When passing object methods as callbacks, for instance to setTimeout, there’s a known problem: "losing this".

In this chapter we’ll see the ways to fix it.

Losing “this”
We’ve already seen examples of losing this. Once a method is passed somewhere separately from the object – this is lost.

Here’s how it may happen with setTimeout:

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(user.sayHi, 1000); // Hello, undefined!
```

As we can see, the output shows not “John” as this.firstName, but undefined!

That’s because setTimeout got the function user.sayHi, separately from the object. The last line can be rewritten as:

```
let f = user.sayHi;
setTimeout(f, 1000); // lost user context
```

The method setTimeout in-browser is a little special: it sets this=window for the function call (for Node.js, this becomes the timer object

The task is quite typical – we want to pass an object method somewhere else (here – to the scheduler) where it will be called. How to make

Solution 1: a wrapper
The simplest solution is to use a wrapping function:

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(function() {
  user.sayHi(); // Hello, John!
}, 1000);
```

Now it works, because it receives user from the outer lexical environment, and then calls the method normally.

The same, but shorter:

```
setTimeout(() => user.sayHi(), 1000); // Hello, John!
```

Looks fine, but a slight vulnerability appears in our code structure.

What if before setTimeout triggers (there’s one second delay!) user changes value? Then, suddenly, it will call the wrong object!

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(() => user.sayHi(), 1000);

// ...the value of user changes within 1 second
user = {
```

```
  sayHi() { alert("Another user in setTimeout!"); }
};

// Another user in setTimeout!
```

The next solution guarantees that such thing won’t happen.

Solution 2: bind for details :
https://javascript.info/bind

40.Arrow functions revisited
Let’s revisit arrow functions.

Arrow functions are not just a “shorthand” for writing small stuff. They have some very specific and useful features.

JavaScript is full of situations where we need to write a small function that’s executed somewhere else.

For instance:

arr.forEach(func) – func is executed by forEach for every array item.
setTimeout(func) – func is executed by the built-in scheduler.
…there are more.
It’s in the very spirit of JavaScript to create a function and pass it somewhere.

And in such functions we usually don’t want to leave the current context. That’s where arrow functions come in handy.

Arrow functions have no “this”
As we remember from the chapter Object methods, "this", arrow functions do not have this. If this is accessed, it is taken from the outside

For instance, we can use it to iterate inside an object method:

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(
      student => alert(this.title + ': ' + student)
    );
  }
};

group.showList();
```

Here in forEach, the arrow function is used, so this.title in it is exactly the same as in the outer method showList. That is: group.title.

If we used a “regular” function, there would be an error:

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(function(student) {
      // Error: Cannot read property 'title' of undefined
      alert(this.title + ': ' + student);
    });
  }
};

group.showList();
```

The error occurs because forEach runs functions with this=undefined by default, so the attempt to access undefined.title is made.

That doesn’t affect arrow functions, because they just don’t have this.

Arrow functions can’t run with new
Not having this naturally means another limitation: arrow functions can’t be used as constructors. They can’t be called with new.

Arrow functions VS bind
There’s a subtle difference between an arrow function => and a regular function called with .bind(this):

.bind(this) creates a “bound version” of the function.
The arrow => doesn’t create any binding. The function simply doesn’t have this. The lookup of this is made exactly the same way as a regula
Arrows have no “arguments”
Arrow functions also have no arguments variable.

That’s great for decorators, when we need to forward a call with the current this and arguments.

For instance, defer(f, ms) gets a function and returns a wrapper around it that delays the call by ms milliseconds:

```
function defer(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(who) {
  alert('Hello, ' + who);
}

let sayHiDeferred = defer(sayHi, 2000);
sayHiDeferred("John"); // Hello, John after 2 seconds
```

The same without an arrow function would look like:

```
function defer(f, ms) {
  return function(...args) {
    let ctx = this;
    setTimeout(function() {
      return f.apply(ctx, args);
    }, ms);
  };
}
```

Here we had to create additional variables args and ctx so that the function inside setTimeout could take them.

Summary
Arrow functions:

Do not have this
Do not have arguments
Can’t be called with new
They also don’t have super, but we didn’t study it yet. We will on the chapter Class inheritance
That’s because they are meant for short pieces of code that do not have their own “context”, but rather work in the current one. And they r

41.Property flags and descriptors for details :
https://javascript.info/property-descriptors

42.Property getters and setters
There are two kinds of object properties.

The first kind is data properties. We already know how to work with them. All properties that we’ve been using until now were data properti

The second type of property is something new. It’s an accessor property. They are essentially functions that execute on getting and setting

Getters and setters
Accessor properties are represented by “getter” and “setter” methods. In an object literal they are denoted by get and set:

```
let obj = {
  get propName() {
    // getter, the code executed on getting obj.propName
  },

  set propName(value) {
    // setter, the code executed on setting obj.propName = value
  }
};
```

The getter works when obj.propName is read, the setter – when it is assigned.

For instance, we have a user object with name and surname:

```
let user = {
  name: "John",
  surname: "Smith"
};
```

Now we want to add a fullName property, that should be "John Smith". Of course, we don’t want to copy-paste existing information, so we can

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

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

alert(user.fullName); // John Smith
```

From the outside, an accessor property looks like a regular one. That’s the idea of accessor properties. We don’t call user.fullName as a f

As of now, fullName has only a getter. If we attempt to assign user.fullName=, there will be an error:

```
let user = {
  get fullName() {
    return `...`;
  }
};

user.fullName = "Test"; // Error (property has only a getter)
```

Let’s fix it by adding a setter for user.fullName:

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

  get fullName() {
    return `${this.name} ${this.surname}`;
  },

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  }
};

// set fullName is executed with the given value.
user.fullName = "Alice Cooper";

alert(user.name); // Alice
alert(user.surname); // Cooper
```

As the result, we have a “virtual” property fullName. It is readable and writable.

Accessor descriptors
Descriptors for accessor properties are different from those for data properties.

For accessor properties, there is no value or writable, but instead there are get and set functions.

That is, an accessor descriptor may have:

get – a function without arguments, that works when a property is read,
set – a function with one argument, that is called when the property is set,
enumerable – same as for data properties,
configurable – same as for data properties.
For instance, to create an accessor fullName with defineProperty, we can pass a descriptor with get and set:

```
let user = {
  name: "John",
  surname: "Smith"
};

Object.defineProperty(user, 'fullName', {
  get() {
    return `${this.name} ${this.surname}`;
  },

  set(value) {
    [this.name, this.surname] = value.split(" ");
  }
});

alert(user.fullName); // John Smith

for(let key in user) alert(key); // name, surname
```

Please note that a property can be either an accessor (has get/set methods) or a data property (has a value), not both.

If we try to supply both get and value in the same descriptor, there will be an error:

```
// Error: Invalid property descriptor.
Object.defineProperty({}, 'prop', {
  get() {
    return 1
  },

  value: 2
});
```

Smarter getters/setters
Getters/setters can be used as wrappers over “real” property values to gain more control over operations with them.

For instance, if we want to forbid too short names for user, we can have a setter name and keep the value in a separate property _name:

```
let user = {
  get name() {
    return this._name;
  },

  set name(value) {
    if (value.length < 4) {
      alert("Name is too short, need at least 4 characters");
      return;
    }
    this._name = value;
  }
};

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

user.name = ""; // Name is too short...
```

So, the name is stored in _name property, and the access is done via getter and setter.

Technically, external code is able to access the name directly by using user._name. But there is a widely known convention that properties

Using for compatibility
One of the great uses of accessors is that they allow to take control over a “regular” data property at any moment by replacing it with a g

Imagine we started implementing user objects using data properties name and age:

```
function User(name, age) {
  this.name = name;
  this.age = age;
}

let john = new User("John", 25);

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

…But sooner or later, things may change. Instead of age we may decide to store birthday, because it’s more precise and convenient:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;
}
```

```
let john = new User("John", new Date(1992, 6, 1));
```

Now what to do with the old code that still uses age property?

We can try to find all such places and fix them, but that takes time and can be hard to do if that code is used by many other people. And b

Let’s keep it.

Adding a getter for age solves the problem:

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

  // age is calculated from the current date and birthday
  Object.defineProperty(this, "age", {
    get() {
      let todayYear = new Date().getFullYear();
      return todayYear - this.birthday.getFullYear();
    }
  });
}

let john = new User("John", new Date(1992, 6, 1));

alert( john.birthday ); // birthday is available
alert( john.age );      // ...as well as the age
```

Now the old code works too and we’ve got a nice additional property.

45.*Prototypal inheritance for details :
https://javascript.info/prototype-inheritance

46.F.prototype for details :
https://javascript.info/function-prototype

47.*Native prototypes for details :
https://javascript.info/native-prototypes

48.Prototype methods, objects without __proto__ for details :
https://javascript.info/prototype-methods

49:* class has mane chapters for details :
https://javascript.info/classes

50.Error handling, "try...catch"

No matter how great we are at programming, sometimes our scripts have errors. They may occur because of our mistakes, an unexpected user in

Usually, a script “dies” (immediately stops) in case of an error, printing it to console.

But there’s a syntax construct try...catch that allows us to “catch” errors so the script can, instead of dying, do something more reasonab

The “try…catch” syntax

The try...catch construct has two main blocks: try, and then catch:

```
try {

  // code...

} catch (err) {

  // error handling

}
```

It works like this:

First, the code in try {...} is executed.
If there were no errors, then catch (err) is ignored: the execution reaches the end of try and goes on, skipping catch.
If an error occurs, then the try execution is stopped, and control flows to the beginning of catch (err). The err variable (we can use any

An errorless example: shows alert (1) and (2):

```
try {

  alert('Start of try runs');  // (1) <--

  // ...no errors here

  alert('End of try runs');   // (2) <--

} catch (err) {

  alert('Catch is ignored, because there are no errors'); // (3)

}
```

An example with an error: shows (1) and (3):

```
try {

  alert('Start of try runs');  // (1) <--

  lalala; // error, variable is not defined!

  alert('End of try (never reached)');  // (2)
```

```
} catch (err) {

  alert(`Error has occurred!`); // (3) <--

}
```

try...catch only works for runtime errors
For try...catch to work, the code must be runnable. In other words, it should be valid JavaScript.

It won’t work if the code is syntactically wrong, for instance it has unmatched curly braces:

```
try {
  {{{{{{{{{{{{
} catch (err) {
  alert("The engine can't understand this code, it's invalid");
}
```

The JavaScript engine first reads the code, and then runs it. The errors that occur on the reading phase are called “parse-time” errors and

So, try...catch can only handle errors that occur in valid code. Such errors are called “runtime errors” or, sometimes, “exceptions”.

try...catch works synchronously
If an exception happens in “scheduled” code, like in setTimeout, then try...catch won’t catch it:

```
try {
  setTimeout(function() {
    noSuchVariable; // script will die here
  }, 1000);
} catch (err) {
  alert( "won't work" );
}
```

That’s because the function itself is executed later, when the engine has already left the try...catch construct.

To catch an exception inside a scheduled function, try...catch must be inside that function:

```
setTimeout(function() {
  try {
    noSuchVariable; // try...catch handles the error!
  } catch {
    alert( "error is caught here!" );
  }
}, 1000);
```

Error object
When an error occurs, JavaScript generates an object containing the details about it. The object is then passed as an argument to catch:

```
try {
  // ...
} catch (err) { // <-- the "error object", could use another word instead of err
  // ...
}
```

For all built-in errors, the error object has two main properties:

name
Error name. For instance, for an undefined variable that’s "ReferenceError".
message
Textual message about error details.
There are other non-standard properties available in most environments. One of most widely used and supported is:

stack
Current call stack: a string with information about the sequence of nested calls that led to the error. Used for debugging purposes.
For instance:

```
try {
  lalala; // error, variable is not defined!
} catch (err) {
  alert(err.name); // ReferenceError
  alert(err.message); // lalala is not defined
  alert(err.stack); // ReferenceError: lalala is not defined at (...call stack)

  // Can also show an error as a whole
  // The error is converted to string as "name: message"
  alert(err); // ReferenceError: lalala is not defined
}
```

Optional “catch” binding
A recent addition
This is a recent addition to the language. Old browsers may need polyfills.
If we don’t need error details, catch may omit it:

```
try {
  // ...
} catch { // <-- without (err)
  // ...
}
```

Using “try…catch”
Let’s explore a real-life use case of try...catch.

As we already know, JavaScript supports the JSON.parse(str) method to read JSON-encoded values.

Usually it’s used to decode data received over the network, from the server or another source.

We receive it and call JSON.parse like this:

```
let json = '{"name":"John", "age": 30}'; // data from the server
```

```
let user = JSON.parse(json); // convert the text representation to JS object

// now user is an object with properties from the string
alert( user.name ); // John
alert( user.age );  // 30
```

You can find more detailed information about JSON in the JSON methods, toJSON chapter.

If json is malformed, JSON.parse generates an error, so the script “dies”.

Should we be satisfied with that? Of course not!

This way, if something’s wrong with the data, the visitor will never know that (unless they open the developer console). And people really

Let’s use try...catch to handle the error:

```
let json = "{ bad json }";

try {

  let user = JSON.parse(json); // <-- when an error occurs...
  alert( user.name ); // doesn't work

} catch (err) {
  // ...the execution jumps here
  alert( "Our apologies, the data has errors, we'll try to request it one more time." );
  alert( err.name );
  alert( err.message );
}
```

Here we use the catch block only to show the message, but we can do much more: send a new network request, suggest an alternative to the vi

Throwing our own errors

What if json is syntactically correct, but doesn’t have a required name property?

Like this:

```
let json = '{ "age": 30 }'; // incomplete data

try {

  let user = JSON.parse(json); // <-- no errors
  alert( user.name ); // no name!

} catch (err) {
  alert( "doesn't execute" );
}
```

Here JSON.parse runs normally, but the absence of name is actually an error for us.

To unify error handling, we’ll use the throw operator.

“Throw” operator

The throw operator generates an error.

The syntax is:

```
throw
```

Technically, we can use anything as an error object. That may be even a primitive, like a number or a string, but it’s better to use object

JavaScript has many built-in constructors for standard errors: Error, SyntaxError, ReferenceError, TypeError and others. We can use them to

Their syntax is:

```
let error = new Error(message);
// or
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

For built-in errors (not for any objects, just for errors), the name property is exactly the name of the constructor. And message is taken

For instance:

```
let error = new Error("Things happen o_O");

alert(error.name); // Error
alert(error.message); // Things happen o_O
```

Let’s see what kind of error JSON.parse generates:

```
try {
  JSON.parse("{ bad json o_O }");
} catch (err) {
  alert(err.name); // SyntaxError
  alert(err.message); // Unexpected token b in JSON at position 2
}
```

As we can see, that’s a SyntaxError.

And in our case, the absence of name is an error, as users must have a name.

So let’s throw it:

```
let json = '{ "age": 30 }'; // incomplete data

try {
```

```
  let user = JSON.parse(json); // <-- no errors

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name"); // (*)
  }

  alert( user.name );

} catch (err) {
  alert( "JSON Error: " + err.message ); // JSON Error: Incomplete data: no name
}
```

In the line (*), the throw operator generates a SyntaxError with the given message, the same way as JavaScript would generate it itself. Th

Now catch became a single place for all error handling: both for JSON.parse and other cases.

Rethrowing

In the example above we use try...catch to handle incorrect data. But is it possible that another unexpected error occurs within the try {.

For example:

```
let json = '{ "age": 30 }'; // incomplete data

try {
  user = JSON.parse(json); // <-- forgot to put "let" before user

  // ...
} catch (err) {
  alert("JSON Error: " + err); // JSON Error: ReferenceError: user is not defined
  // (no JSON Error actually)
}
```

Of course, everything’s possible! Programmers do make mistakes. Even in open-source utilities used by millions for decades – suddenly a bug

In our case, try...catch is placed to catch “incorrect data” errors. But by its nature, catch gets all errors from try. Here it gets an une

To avoid such problems, we can employ the “rethrowing” technique. The rule is simple:

Catch should only process errors that it knows and “rethrow” all others.

The “rethrowing” technique can be explained in more detail as:

Catch gets all errors.
In the catch (err) {...} block we analyze the error object err.
If we don’t know how to handle it, we do throw err.
Usually, we can check the error type using the instanceof operator:

```
try {
  user = { /*...*/ };
} catch (err) {
  if (err instanceof ReferenceError) {
    alert('ReferenceError'); // "ReferenceError" for accessing an undefined variable
  }
}
```

We can also get the error class name from err.name property. All native errors have it. Another option is to read err.constructor.name.

In the code below, we use rethrowing so that catch only handles SyntaxError:

```
let json = '{ "age": 30 }'; // incomplete data
try {

  let user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name");
  }

  blabla(); // unexpected error

  alert( user.name );

} catch (err) {

  if (err instanceof SyntaxError) {
    alert( "JSON Error: " + err.message );
  } else {
    throw err; // rethrow (*)
  }

}
```

The error throwing on line (*) from inside catch block “falls out” of try...catch and can be either caught by an outer try...catch construc

So the catch block actually handles only errors that it knows how to deal with and “skips” all others.

The example below demonstrates how such errors can be caught by one more level of try...catch:

```
function readData() {
  let json = '{ "age": 30 }';

  try {
    // ...
    blabla(); // error!
  } catch (err) {
```

```
    // ...
    if (!(err instanceof SyntaxError)) {
      throw err; // rethrow (don't know how to deal with it)
    }
  }
}

try {
  readData();
} catch (err) {
  alert( "External catch got: " + err ); // caught it!
}
```

Here readData only knows how to handle SyntaxError, while the outer try...catch knows how to handle everything.

try…catch…finally
Wait, that’s not all.

The try...catch construct may have one more code clause: finally.

If it exists, it runs in all cases:

after try, if there were no errors,
after catch, if there were errors.
The extended syntax looks like this:

```
try {
   ... try to execute the code ...
} catch (err) {
   ... handle errors ...
} finally {
   ... execute always ...
}
```

Try running this code:

```
try {
  alert( 'try' );
  if (confirm('Make an error?')) BAD_CODE();
} catch (err) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

The code has two ways of execution:

If you answer “Yes” to “Make an error?”, then try -> catch -> finally.
If you say “No”, then try -> finally.
The finally clause is often used when we start doing something and want to finalize it in any case of outcome.

For instance, we want to measure the time that a Fibonacci numbers function fib(n) takes. Naturally, we can start measuring before it runs

The finally clause is a great place to finish the measurements no matter what.

Here finally guarantees that the time will be measured correctly in both situations – in case of a successful execution of fib and in case

```
let num = +prompt("Enter a positive integer number?", 35)

let diff, result;

function fib(n) {
  if (n < 0 || Math.trunc(n) != n) {
    throw new Error("Must not be negative, and also an integer.");
  }
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

let start = Date.now();

try {
  result = fib(num);
} catch (err) {
  result = 0;
} finally {
  diff = Date.now() - start;
}

alert(result || "error occurred");

alert( `execution took ${diff}ms` );
```

You can check by running the code with entering 35 into prompt – it executes normally, finally after try. And then enter -1 – there will be

In other words, the function may finish with return or throw, that doesn’t matter. The finally clause executes in both cases.

Variables are local inside try...catch...finally
Please note that result and diff variables in the code above are declared before try...catch.

Otherwise, if we declared let in try block, it would only be visible inside of it.

finally and return
The finally clause works for any exit from try...catch. That includes an explicit return.

In the example below, there’s a return in try. In this case, finally is executed just before the control returns to the outer code.

```
function func() {

  try {
    return 1;

  } catch (err) {
    /* ... */
  } finally {
    alert( 'finally' );
  }
}

alert( func() ); // first works alert from finally, and then this one
```

try...finally

The try...finally construct, without catch clause, is also useful. We apply it when we don’t want to handle errors here (let them fall thro

```
function func() {
  // start doing something that needs completion (like measurements)
  try {
    // ...
  } finally {
    // complete that thing even if all dies
  }
}
```

In the code above, an error inside try always falls out, because there’s no catch. But finally works before the execution flow leaves the f

Global catch

Environment-specific

The information from this section is not a part of the core JavaScript.

Let’s imagine we’ve got a fatal error outside of try...catch, and the script died. Like a programming error or some other terrible thing.

Is there a way to react on such occurrences? We may want to log the error, show something to the user (normally they don’t see error messag

There is none in the specification, but environments usually provide it, because it’s really useful. For instance, Node.js has process.on("

The syntax:

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

message

Error message.

url

URL of the script where error happened.

line, col

Line and column numbers where error happened.

error

Error object.

For instance:

The role of the global handler window.onerror is usually not to recover the script execution – that’s probably impossible in case of progra

There are also web-services that provide error-logging for such cases, like https://errorception.com or http://www.muscula.com.

They work like this:

We register at the service and get a piece of JS (or a script URL) from them to insert on pages.
That JS script sets a custom window.onerror function.
When an error occurs, it sends a network request about it to the service.
We can log in to the service web interface and see errors.

Summary

The try...catch construct allows to handle runtime errors. It literally allows to “try” running the code and “catch” errors that may occur

The syntax is:

```
try {
  // run this code
} catch (err) {
  // if an error happened, then jump here
  // err is the error object
} finally {
  // do in any case after try/catch
}
```

There may be no catch section or no finally, so shorter constructs try...catch and try...finally are also valid.

Error objects have following properties:

message – the human-readable error message.
name – the string with error name (error constructor name).
stack (non-standard, but well-supported) – the stack at the moment of error creation.

If an error object is not needed, we can omit it by using catch { instead of catch (err) {.

We can also generate our own errors using the throw operator. Technically, the argument of throw can be anything, but usually it’s an error

Rethrowing is a very important pattern of error handling: a catch block usually expects and knows how to handle the particular error type,

Even if we don’t have try...catch, most environments allow us to setup a “global” error handler to catch errors that “fall out”. In-browser

51.Custom errors, extending Error

When we develop something, we often need our own error classes to reflect specific things that may go wrong in our tasks. For errors in net

Our errors should support basic error properties like message, name and, preferably, stack. But they also may have other properties of thei

JavaScript allows to use throw with any argument, so technically our custom error classes don't need to inherit from Error. But if we inher

As the application grows, our own errors naturally form a hierarchy. For instance, HttpTimeoutError may inherit from HttpError, and so on.

Extending Error
As an example, let's consider a function readUser(json) that should read JSON with user data.

Here's an example of how a valid json may look:

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

Internally, we'll use JSON.parse. If it receives malformed json, then it throws SyntaxError. But even if json is syntactically correct, tha

Our function readUser(json) will not only read JSON, but check ("validate") the data. If there are no required fields, or the format is wro

Our ValidationError class should inherit from the Error class.

The Error class is built-in, but here's its approximate code so we can understand what we're extending:

```
// The "pseudocode" for the built-in Error class defined by JavaScript itself
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (different names for different built-in error classes)
    this.stack = ; // non-standard, but most environments support it
  }
}
```

Now let's inherit ValidationError from it and try it in action:

```
class ValidationError extends Error {
  constructor(message) {
    super(message); // (1)
    this.name = "ValidationError"; // (2)
  }
}

function test() {
  throw new ValidationError("Whoops!");
}

try {
  test();
} catch(err) {
  alert(err.message); // Whoops!
  alert(err.name); // ValidationError
  alert(err.stack); // a list of nested calls with line numbers for each
}
```

Please note: in the line (1) we call the parent constructor. JavaScript requires us to call super in the child constructor, so that's oblig

The parent constructor also sets the name property to "Error", so in the line (2) we reset it to the right value.

Let's try to use it in readUser(json):

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

// Usage
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new ValidationError("No field: age");
  }
  if (!user.name) {
    throw new ValidationError("No field: name");
  }

  return user;
}

// Working example with try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Invalid data: No field: name
  } else if (err instanceof SyntaxError) { // (*)
    alert("JSON Syntax Error: " + err.message);
  } else {
    throw err; // unknown error, rethrow it (**)
  }
}
```

The try..catch block in the code above handles both our ValidationError and the built-in SyntaxError from JSON.parse.

Please take a look at how we use instanceof to check for the specific error type in the line (*).

We could also look at err.name, like this:

```
// ...
// instead of (err instanceof SyntaxError)
} else if (err.name == "SyntaxError") { // (*)
// ...
```

The instanceof version is much better, because in the future we are going to extend ValidationError, make subtypes of it, like PropertyRequ

Also it’s important that if catch meets an unknown error, then it rethrows it in the line (**). The catch block only knows how to handle va

Further inheritance

The ValidationError class is very generic. Many things may go wrong. The property may be absent or it may be in a wrong format (like a stri

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.name = "PropertyRequiredError";
    this.property = property;
  }
}

// Usage
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new PropertyRequiredError("age");
  }
  if (!user.name) {
    throw new PropertyRequiredError("name");
  }

  return user;
}

// Working example with try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Invalid data: No property: name
    alert(err.name); // PropertyRequiredError
    alert(err.property); // name
  } else if (err instanceof SyntaxError) {
    alert("JSON Syntax Error: " + err.message);
  } else {
    throw err; // unknown error, rethrow it
  }
}
```

The new class PropertyRequiredError is easy to use: we only need to pass the property name: new PropertyRequiredError(property). The human-

Please note that this.name in PropertyRequiredError constructor is again assigned manually. That may become a bit tedious – to assign this.

Let’s call it MyError.

Here’s the code with MyError and other custom error classes, simplified:

```
class MyError extends Error {
  constructor(message) {
    super(message);
    this.name = this.constructor.name;
  }
}

class ValidationError extends MyError { }

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.property = property;
  }
}

// name is correct
alert( new PropertyRequiredError("field").name ); // PropertyRequiredError
```

Now custom errors are much shorter, especially ValidationError, as we got rid of the "this.name = ..." line in the constructor.

Wrapping exceptions

The purpose of the function readUser in the code above is “to read the user data”. There may occur different kinds of errors in the process

The code which calls readUser should handle these errors. Right now it uses multiple ifs in the catch block, that check the class and handl

The scheme is like this:

```
try {
  ...
  readUser()  // the potential error source
  ...
} catch (err) {
  if (err instanceof ValidationError) {
    // handle validation errors
  } else if (err instanceof SyntaxError) {
    // handle syntax errors
  } else {
    throw err; // unknown error, rethrow it
  }
}
```

In the code above we can see two types of errors, but there can be more.

If the readUser function generates several kinds of errors, then we should ask ourselves: do we really want to check for all error types on

Often the answer is “No”: we’d like to be “one level above all that”. We just want to know if there was a “data reading error” – why exactl

The technique that we describe here is called “wrapping exceptions”.

We’ll make a new class ReadError to represent a generic “data reading” error.
The function readUser will catch data reading errors that occur inside it, such as ValidationError and SyntaxError, and generate a ReadErro
The ReadError object will keep the reference to the original error in its cause property.
Then the code that calls readUser will only have to check for ReadError, not for every kind of data reading errors. And if it needs more de

Here’s the code that defines ReadError and demonstrates its use in readUser and try..catch:

```
class ReadError extends Error {
  constructor(message, cause) {
    super(message);
    this.cause = cause;
    this.name = 'ReadError';
  }
}

class ValidationError extends Error { /*...*/ }
class PropertyRequiredError extends ValidationError { /* ... */ }

function validateUser(user) {
  if (!user.age) {
    throw new PropertyRequiredError("age");
  }

  if (!user.name) {
    throw new PropertyRequiredError("name");
  }
}

function readUser(json) {
  let user;

  try {
    user = JSON.parse(json);
  } catch (err) {
    if (err instanceof SyntaxError) {
      throw new ReadError("Syntax Error", err);
    } else {
      throw err;
    }
  }

  try {
    validateUser(user);
  } catch (err) {
    if (err instanceof ValidationError) {
      throw new ReadError("Validation Error", err);
    } else {
      throw err;
    }
  }

}

try {
  readUser('{bad json}');
} catch (e) {
  if (e instanceof ReadError) {
    alert(e);
    // Original error: SyntaxError: Unexpected token b in JSON at position 1
    alert("Original error: " + e.cause);
  } else {
    throw e;
  }
}
```

In the code above, readUser works exactly as described – catches syntax and validation errors and throws ReadError errors instead (unknown

So the outer code checks instanceof ReadError and that’s it. No need to list all possible error types.

The approach is called “wrapping exceptions”, because we take “low level” exceptions and “wrap” them into ReadError that is more abstract.

52.