1. We can put any type in a variable. Programming languages that allow such things, such as JavaScript, are called “dynamically typed”, meaning that there exist data types, but variables are not bound to any of them.
2. Numbers: there are so-called “special numeric values” which also belong to this data type: Infinity, -Infinity and NaN
3. NaN represents a computational error. It is a result of an incorrect or an undefined mathematical operation
4. For most purposes ±(253-1) range is quite enough, but sometimes we need the entire range of really big integers, e.g. for cryptography or microsecond-precision timestamps.
5. BigInt type was recently added to the language to represent integers of arbitrary length.
6. Backticks are “extended functionality” quotes. They allow us to embed variables and expressions into a string by wrapping them in ${…}
7. 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.
8. null is not a “reference to a non-existing object” or a “null pointer” like in some other languages.
9. It’s just a special value which represents “nothing”, “empty” or “value unknown”.
10. The meaning of undefined is “value is not assigned. If a variable is declared, but not assigned, then its value is undefined
11. objects are used to store collections of data and more complex entities.
12. The symbol type is used to create unique identifiers for objects.
13. Symbols is a build in object whose constructor returns symbol primitive – also called symbol value – that’s guaranteed to be unique. Symbols are often used to ad unique properties to objects that won’t collide with keys.
14. Every Symbol() call is guaranteed to return a unique symbol.
15. The result of typeof null is "object". That’s an officially recognized error in typeof, coming from very early days of JavaScript and kept for compatibility. Definitely, null is not an object. It is a special value with a separate type of its own. The behavior of typeof is wrong here
16. The result of typeof alert is "function", because alert is a function. We’ll study functions in the next chapters where we’ll also see that there’s no special “function” type in JavaScript. Functions belong to the object type. But typeof treats them differently, returning "function". That also comes from the early days of JavaScript.

**Type Conversions**

1. alert automatically converts any value to a string
2. String conversion: value = String(value)
3. Number Conversion: let num = Number(str) OR +num;

|  |  |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true and false | 1 and 0 |

**Basic Operators and Maths**

1. An operand – is what operators are applied to example 5 \* 2 5and 2 is operand.
2. An operator is unary if it has a single operand. Example: -x
3. An operator is binary if it has two operands example: y – x;
4. The binary + is the only operator that supports strings in such a way. Other arithmetic operators work only with numbers and always convert their operands to numbers.
5. Increment/decrement can only be applied to variables. Trying to use it on a value like 5++ will give an error.

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

1. Bitwise operators treat arguments as 32-bit integer numbers and work on the level of their binary representation.

These operators are not JavaScript-specific. They are supported in most programming languages.

The list of operators:

* AND ( & )
* OR ( | )
* XOR ( ^ )
* NOT ( ~ )
* LEFT SHIFT ( << )
* RIGHT SHIFT ( >> )
* ZERO-FILL RIGHT SHIFT ( >>> )

1. These operators are used very rarely, when we need to fiddle with numbers on the very lowest (bitwise) level.

**Comparision operator**

* 1. All comparison operators return a boolean value:
  2. When comparing values of different types, JavaScript converts the values to numbers. Example: alert( '2' > 1 ); // true, string '2' becomes a number 2
  3. alert( '' == false ); // true
  4. **A strict equality operator === checks the equality without type conversion.**
  5. In other words, if a and b are of different types, then a === b immediately returns false without an attempt to convert them.
  6. alert( null === undefined ); // false
  7. alert( null == undefined ); // true

**Conditional Branching: If, “?”**

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

**Logical Operators**

* + 1. There are four logical operators in JavaScript: || (OR), && (AND), ! (NOT), ?? (Nullish Coalescing).
    2. OR || If any of its arguments are true, it returns true, otherwise it returns false
    3. Another feature of OR || operator is the so-called “short-circuit” evaluation.
    4. It means that || processes its arguments until the first truthy value is reached, and then the value is returned immediately, without even touching the other argument
    5. AND returns true if both operands are truthy and false otherwise
    6. OR (||) finds the first truthy value and and AND(&&) finds the first falsy value.
    7. Nullish coalescing operator (??).

he result of a ?? b is:

if a is defined, then a,if a isn’t defined, then b

* 1. In other words, ?? returns the first argument if it’s not null/undefined. Otherwise, the second one.
  2. The nullish coalescing operator ?? provides a short way to choose the first “defined” value from a list.

**Loops: While and for**

* + 1. While loop: While the condition is truthy, the code from the loop body is executed.
    2. Do while: The loop will first execute the body, then check the condition, and, while it’s truthy, execute it again and again.
    3. loop to execute **at least once** regardless of the condition being truthy.
    4. To make an “infinite” loop, usually the while(true) construct is used. Such a loop, just like any other, can be stopped with the break directive.
    5. If we don’t want to do anything in the current iteration and would like to forward to the next one, we can use the continue directive.
    6. break/continue support labels before the loop. A label is the only way for break/continue to escape a nested loop to go to an outer one.
    7. Type coercion is **the automatic or implicit conversion of values from one data type to another** (such as strings to numbers).

**Functions**

1. Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.
2. Function Declaration: function hello() {  
    Code//  
   }
3. Local variable and global/ Outer Variable
4. Variable Shadowing
5. Return : A function can return a value back into the calling code as the result.
6. Function Expression:
7. Let sayHi = function(){  
    Code//  
   }
8. Let’s reiterate: no matter how the function is created, a function is a value.
9. Parameter: A parameter is the variable listed inside the parentheses in the function declaration (it’s a declaration time term).
10. An argument is the value that is passed to the function when it is called (it’s a call time term).
11. Function expression can be anonymous while, function declaration must have name.
12. **A Function Expression is created when the execution reaches it and is usable only from that moment. -** Once the execution flow passes to the right side of the assignment let sum = function… – here we go, the function is created and can be used (assigned, called, etc. ) from now on.
13. **Function Declaration can be called earlier than it is defined. – A**  global Function Declaration is visible in the whole script, no matter where it is.

That’s due to internal algorithms. When JavaScript prepares to run the script, it first looks for global Function Declarations in it and creates the functions. We can think of it as an “initialization stage”.

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

## [Summary](https://javascript.info/function-expressions" \l "summary)

* Functions are values. They can be assigned, copied or declared in any place of the code.
* If the function is declared as a separate statement in the main code flow, that’s called a “Function Declaration”.
* If the function is created as a part of an expression, it’s called a “Function Expression”.
* Function Declarations are processed before the code block is executed. They are visible everywhere in the block.
* Function Expressions are created when the execution flow reaches them.

In most cases when we need to declare a function, a Function Declaration is preferable, because it is visible prior to the declaration itself. That gives us more flexibility in code organization, and is usually more readable.

So we should use a Function Expression only when a Function Declaration is not fit for the task. We’ve seen a couple of examples of that in this chapter, and will see more in the future.

**Arrow Function Basics**

* 1. There’s another very simple and concise syntax for creating functions, that’s often better than Function Expressions.
  2. It’s called “arrow functions
  3. let func = (arg1, arg2, ..., argN) => expression;

## [Summary](https://javascript.info/arrow-functions-basics#summary)

Arrow functions are handy for simple actions, especially for one-liners. They come in two flavors:

1. Without curly braces: (...args) => expression – the right side is an expression: the function evaluates it and returns the result. Parentheses can be omitted, if there’s only a single argument, e.g. n => n\*2.
2. With curly braces: (...args) => { body } – brackets allow us to write multiple statements inside the function, but we need an explicit return to return something.

Transpilers: A [transpiler](https://en.wikipedia.org/wiki/Source-to-source_compiler) is a special piece of software that translates source code to another source code. It can parse (“read and understand”) modern code and rewrite it using older syntax constructs, so that it’ll also work in outdated engines.

height ?? 100 into (height !== undefined && height !== null) ? height : 100

transpiler (if using modern syntax or operators) and polyfills (to add functions that may be missing). They’ll ensure that the code works

**Objects**

1. objects are used to store keyed collections of various data and more complex entities.
2. Object Constructor and Object literal

### Key Differences:

1. **Syntax**:
   * Object Constructor: Requires a function and new keyword.
   * Object Literal: Directly defines an object using {} with key-value pairs.
2. **Reusability**:
   * Object Constructor: Allows creating multiple instances (using new).
   * Object Literal: Typically used for defining a single object.
3. **Performance**:
   * Object Constructor: More efficient if you need to create multiple objects of the same type, as you can reuse the constructor.
   * Object Literal: Suitable for a single, one-off object.
4. **Inheritance**:
   * Object Constructor: Can inherit from a prototype, allowing for shared methods and properties.
   * Object Literal: Doesn't naturally support inheritance (though inheritance can be achieved through other techniques like Object.create).
5. A property has a key (also known as “name” or “identifier”) before the colon ":" and a value to the right of it.
6. To remove a property, we can use the delete operator:
7. We can also use multiword property names, but then they must be quoted:
8. For multiword properties, the dot access doesn’t work:
9. alert(user["likes birds"]);

 \_\_proto\_\_ refers to the prototype of an object and is part of the prototype chain.

 It allows objects to inherit properties and methods from other objects.

1.  While it works, it's generally recommended to use Object.getPrototypeOf() and Object.setPrototypeOf() instead, as \_\_proto\_\_ is now considered somewhat outdated.
2. The in operator in JavaScript is used to check if a specified property exists in an object or in an array (or other iterable). It checks the presence of the property in the object's prototype chain as well, not just the object's own properties.
3. Syntax: property in object
4. Why does the in operator exist? Isn’t it enough to compare against undefined?
5. Well, most of the time the comparison with undefined works fine. But there’s a special case when it fails, but "in" works correctly.
6. It’s when an object property exists, but stores undefined

**For in Loop**

1. To walk over all keys of an object, there exists a special form of the loop: for..in.

for (key in object) {

// executes the body for each key among object properties

}

1. let user = {
2. name: "saqib",
3. age: 23,
4. JOb: "developer",
5. };
6. for (let key in user) {
7. console.log(key); // Keys
8. console.log(user[key]); // Value of Keys
9. }
10. we could use another variable name here instead of key. For instance, "for (let prop in obj)" is also widely used.
11. The short answer is: “ordered in a special fashion”: integer properties are sorted, others appear in creation order.

## [Summary](https://javascript.info/object#summary)

Objects are associative arrays with several special features.

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

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

To access a property, we can use:

* The dot notation: obj.property.
* Square brackets notation obj["property"]. Square brackets allow 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.
* …And so on.

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

**Object References and copying**

* 1. One of the fundamental differences of objects versus primitives is that objects are stored and copied “by reference”, whereas primitive values: strings, numbers, booleans, etc – are always copied “as a whole value”.
  2. **A variable assigned to an object stores not the object itself, but its “address in memory” – in other words “a reference” to it.**
  3. **When an object variable is copied, the reference is copied, but the object itself is not duplicated.**

let user = { name: "John" };

let admin = user; // copy the reference

* 1. Two objects are equal only if they are the same object.
  2. For instance, here a and b reference the same object, thus they are equal:

let a = {};

let b = a; // copy the reference

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

alert( a === b ); // true

* 1. And here two independent objects are not equal, even though they look alike (both are empty):

let a = {};

let b = {}; // two independent objects

alert( a == b ); // false

* 1. An important side effect of storing objects as references is that an object declared as const can be modified.

**Shallow Copy**

A **shallow copy** of an object means that a new object is created with the same properties as the original object, but the copying is only done at the top level. If the original object contains references to other objects (e.g., nested objects or arrays), the shallow copy will not create new instances of those nested objects. Instead, it will simply copy the reference (i.e., memory address) to the nested objects. This means that changes to nested objects in the shallow copy will also affect the original object and vice versa.

|  |  |  |
| --- | --- | --- |
| Method | Syntax Example | Behavior |
| Object.assign() | const shallowCopy = Object.assign({}, original); | Copies properties from the original object. |
| Spread Operator (...) | const shallowCopy = { ...original }; | Shorthand for copying properties into a new object. |
| Object.create() | const shallowCopy = Object.create(Object.getPrototypeOf(original), Object.getOwnPropertyDescriptors(original)); | Copies properties along with the prototype. |
| for...in Loop | for (let key in original) {...} | Manually copies properties. |
| JSON.parse() & JSON.stringify() | const shallowCopy = JSON.parse(JSON.stringify(original)); | Not recommended for shallow copy, often leads to deep copy. |

The first four methods are generally recommended for creating shallow copies.

**Deeep Copy**

A **deep copy** of an object means creating a new object, including all of its nested objects (or arrays), with all properties being recursively copied. The key difference between a shallow copy and a deep copy is that in a deep copy, **nested objects are cloned entirely**—they do not retain references to the original nested objects. This means that modifying a nested object in the deep copy does not affect the original object, and vice versa.

| **Method** | **Syntax Example** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| JSON.parse() & JSON.stringify() | const deepCopy = JSON.parse(JSON.stringify(original)); | Simple and concise for most cases. | Does not handle special objects (Date, Map, etc.) and functions. |
| Recursive Function | function deepCopy(obj) { ... } | Fully customizable, works for most objects. | More complex to implement, needs handling for special cases. |
| Lodash cloneDeep | const deepCopy = \_.cloneDeep(original); | Handles most edge cases, including circular references. | Requires external library. |
| structuredClone() | const deepCopy = structuredClone(original); | Handles most complex objects, including Date, Map, Set. | Not universally supported. |
| Map/Set Custom Handling | function deepCopyMap(map) { ... } | Works specifically with Map and Set objects. | Requires custom implementation for different types. |

**Garbage Collection**

**garbage collection** (GC) is the process of automatically identifying and reclaiming memory that is no longer in use or reachable by the program. JavaScript uses a **memory management system** where the engine tracks memory allocation for objects, arrays, functions, etc., and automatically frees up memory that is no longer needed.

The primary goal of garbage collection is to optimize memory usage and ensure that the program does not run out of memory. Garbage collection in JavaScript is typically **automatic**, meaning developers do not need to manually free memory.

**Two main garbage collection strategies**:

* **Mark-and-Sweep Algorithm**: JavaScript primarily uses this method. It works by "marking" objects that are reachable, and then "sweeping" (deleting) all unmarked objects.
* **Reference Counting**: This method tracks how many references are pointing to an object and frees objects with zero references. However, it’s rarely used in modern engines due to difficulties handling circular references.

### How Does Garbage Collection Work in JavaScript?

1. **Mark Phase**:
   * The garbage collector begins by **marking** all the reachable objects. It starts from "roots" like global objects, local variables, and function parameters. If an object is reachable from the root, it is marked.
2. **Sweep Phase**:
   * Once the reachable objects are marked, the garbage collector proceeds to sweep through the heap and collects all objects that were not marked (i.e., unreachable objects).
3. **Finalization**:
   * The collected objects are then cleaned up, and the memory they occupy is returned to the memory pool for reuse.

### Common Types of Garbage Collection in JavaScript

1. **Generational Garbage Collection**:
   * This approach is based on the observation that most objects in a program die young (they are created and discarded quickly). As a result, the heap is divided into generations: **young generation** (new objects) and **old generation** (long-lived objects).
   * The garbage collector collects objects from the young generation more frequently than from the old generation.
2. **Incremental and Incremental Mark-Sweep**:
   * JavaScript engines like V8 (used in Chrome and Node.js) perform **incremental garbage collection**, meaning the process is broken into smaller chunks to avoid blocking the main execution thread for too long.
   * **Incremental Mark-Sweep** involves marking reachable objects in small steps and sweeping (collecting) in stages, which helps improve performance during GC.
3. **Lazy Garbage Collection**:
   * This method involves delaying garbage collection until it's absolutely necessary (i.e., when the memory is about to be exhausted).

**Common Causes of Memory Leaks**

**Global Variables**: Storing large objects in global variables that are never cleared can prevent them from being garbage collected.

**Event Listeners**: Not removing event listeners properly can result in memory leaks because the objects involved in the event listener will remain referenced.

**Closures**: Closures can unintentionally retain memory by keeping references to variables from outer scopes.

**Object Methods This**

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

let user = {

name: "John",

  age: 30

};

user.sayHi = function() {

  alert("Hello!");

};

* 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.

1. Technically, it’s also possible to access the object without this, by referencing it via the outer variable:
2. But such code is unreliable. If we decide to copy user to another variable, e.g. admin = user and overwrite user with something else, then it will access the wrong object.

Object Method “this”