Types, Values, and Variables

1 Overview and Definitions

JavaScript types can be divided into two categories: primitive types and object types. JavaScript's primitive types include numbers, strings of text (known as strings), and Boolean truth values (known as booleans). A significant portion of this chapter is dedicated to a detailed explanation of the numeric (§3.2) and string (§3.3) types in JavaScript. Booleans are covered in §3.4. The special JavaScript values null and undefined are primitive values, but they are not numbers, strings, or booleans. Each value is typically considered to be the sole member of its own special type. §3.5 has more about null and undefined. ES6 adds a new special-purpose type, known as Symbol, that enables the definition of language extensions without harming backward compatibility. Symbols are covered briefly in §3.6. Any JavaScript value that is not a number, a string, a boolean, a symbol, null, or unde fined is an object. An object (that is, a member of the type object) is a collection of properties where each property has a name and a value (either a primitive value oranother object). One very special object, the global object, is covered in §3.7, but more general and more detailed coverage of objects is in Chapter 6. An ordinary JavaScript object is an unordered collection of named values. The lan- guage also defines a special kind of object, known as an array, that represents an ordered collection of numbered values. The JavaScript language includes special syn- tax for working with arrays, and arrays have some special behavior that distinguishes them from ordinary objects. Arrays are the subject of Chapter 7. In addition to basic objects and arrays, JavaScript defines a number of other useful object types. A Set object represents a set of values. A Map object represents a map- ping from keys to values. Various "typed array" types facilitate operations on arrays of bytes and other binary data. The RegExp type represents textual patterns and enables sophisticated matching, searching, and replacing operations on strings. The Date type represents dates and times and supports rudimentary date arithmetic. Error and its subtypes represent errors that can arise when executing JavaScript code. All of these types are covered in Chapter 11. JavaScript differs from more static languages in that functions and classes are not just part of the language syntax: they are themselves values that can be manipulated by JavaScript programs. Like any JavaScript value that is not a primitive value, functions and classes are a specialized kind of object. The JavaScript interpreter performs automatic garbage collection for memory man- agement. This means that a JavaScript programmer generally does not need to worry about destruction or deallocation of objects or other values. When a value is no longer reachable—when a program no longer has any way to refer to it—the inter- preter knows it can never be used again and automatically reclaims the memory it was occupying. (JavaScript programmers do sometimes need to take care to ensure that values do not inadvertently remain reachable—and therefore nonreclaimable— longer than necessary.) JavaScript supports an object-oriented programming style. Loosely, this means that rather than having globally defined functions to operate on values of various types, the types themselves define methods for working with values. To sort the elements of an array a, for example, we don't pass a to a sort() function. Instead, we invoke the sort() method of a: a.sort(); // The object-oriented version of sort(a). Method definition is covered in Chapter 9. Technically, it is only JavaScript objects that have methods. But numbers, strings, boolean, and symbol values behave as if they have methods. In JavaScript, null and undefined are the only values that meth- ods cannot be invoked on.

avaScript's object types are mutable and its primitive types are immutable. A value of a mutable type can change: a JavaScript program can change the values of object properties and array elements.

Numbers, booleans, symbols, null, and undefined are immutable—it doesn't even make sense to talk about changing the value of a number, for example. Strings can be thought of as arrays of characters, and you might expect them to be mutable. In JavaScript, however, strings are immutable: you can access the text at any index of a string, but JavaScript provides no way to alter the text of an existing string. The differences between mutable and immutable values are explored further in §3.8. JavaScript liberally converts values from one type to another. If a program expects a string, for example, and you give it a number, it will automatically convert the num- ber to a string for you. And if you use a non-boolean value where a boolean is expected, JavaScript will convert accordingly. The rules for value conversion are explained in §3.9. JavaScript's liberal value conversion rules affect its definition of equality, and the == equality operator performs type conversions as described in §3.9.1. (In practice, however, the == equality operator is deprecated in favor of the strict equality operator ===, which does no type conversions. See §4.9.1 for more about both operators.) Constants and variables allow you to use names to refer to values in your programs. Constants are declared with const and variables are declared with let (or with var in older JavaScript code). JavaScript constants and variables are untyped: declarations do not specify what kind of values will be assigned. Variable declaration and assignment are covered in §3.10. As you can see from this long introduction, this is a wide-ranging chapter that explains many fundamental details about how data is represented and manipulated in JavaScript. We'll begin by diving right in to the details of JavaScript numbers and text.

3.2 Numbers

The Number is a primitive/old data type used for positive or negative integer, float, binary, octal, hexadecimal, and exponential values in JavaScript.

The Number type in JavaScript is double-precision 64 bit binary format like double in C# and Java. It follows the international IEEE 754 standard(The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).).

The first character in a number type must be an integer value, and it must not be enclosed in quotation marks. The following example shows the variables having different types of numbers in JavaScript.

```
<!DOCTYPE html>

<html>

<body>

        <h1>Demo: JavaScript Numbers </h1>


        <p id="p1">Integer: </p>

        <p id="p2">Negative Integer: </p>

        <p id="p3">Float: </p>
```

```
        <p id="p4">Negative Float: </p>
        <p id="p5">Hexadecimal: </p>
        <p id="p6">Exponential: </p>
        <p id="p7">Octal: </p>
        <p id="p8">Binary: </p>

        <script>
            var num1 = 100; // integer
            var num2 = -100; //negative integer

            var num3 = 10.52; // float
            var num4 = -10.52; //negative float

            var num5 = 0xfff; // hexadecimal
            var num6 = 256e-5; // exponential
            var num7 = 030; // octal
            var num8 = 0b0010001; // binary

            document.getElementById("p1").textContent += num1;
            document.getElementById("p2").textContent += num2;
            document.getElementById("p3").textContent += num3;
            document.getElementById("p4").textContent += num4;
            document.getElementById("p5").textContent += num5;
            document.getElementById("p6").textContent += num6;
            document.getElementById("p7").textContent += num7;
            document.getElementById("p8").textContent += num8;
    </script>
</body>
</html>
```

JavaScript's primary numeric type, Number, is used to represent integers and to approximate real numbers. JavaScript represents numbers using the 64-bit floatingpoint format defined by the IEEE 754 standard,1 which means it can represent num- bers as large as ±1.7976931348623157 × 10308 and as small as ±5 × 10−324 . The JavaScript number format allows you to exactly represent all integers between −9,007,199,254,740,992 (−253) and 9,007,199,254,740,992 (253), inclusive. If you use integer values larger than this, you may lose precision in the trailing digits. Note, however, that certain operations in JavaScript (such as array indexing and the bitwise operators described in Chapter 4) are performed with 32-bit integers. If you need to exactly represent larger integers, see §3.2.5. When a number appears directly in a JavaScript program, it's called a numeric literal. JavaScript supports numeric literals in several formats, as described in the following sections. Note that any numeric literal can be preceded by a minus sign (-) to make the number negative.

3.3 Text

The JavaScript type for representing text is the string. A string is an immutable ordered sequence of 16-bit values, each of which typically represents a Unicode char- acter. The length of a string is the number of 16-bit values it contains. JavaScript's strings (and its arrays) use zero-based indexing: the first 16-bit value is at position 0, the second at position 1, and so on. The empty string is the string of length 0. Java-Script does not have a special type that represents a single element of a string. To rep- resent a single 16-bit value, simply use a string that has a length of 1.

Exmple:

1. const string = "The revolution will not be televised.";
console.log(string);

2. const name = "Front ";

const number = 242;

console.log(`${name}${number}`); // "Front 242"

3.4 Boolean Values

Very often, in programming, you will need a data type that can only have one of two values, like

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, JavaScript has a Boolean data type. It can only take the values true or false.

Example:

<script>

```
let x = -0;

document.getElementById("demo").innerHTML = Boolean(x);

</script>
```

2. Everything With a "Value" is True

```
<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML =

"100 is " + Boolean(100) + "<br>" +

"3.14 is " + Boolean(3.14) + "<br>" +

"-15 is " + Boolean(-15) + "<br>" +

"Any (not empty) string is " + Boolean("Hello") + "<br>" +

"Even the string 'false' is " + Boolean('false') + "<br>" +

"Any expression (except zero) is " + Boolean(1 + 7 + 3.14);

</script>
```

A boolean value represents truth or falsehood, on or off, yes or no. There are only two possible values of this type. The reserved words true and false evaluate to these two values. Boolean values are generally the result of comparisons you make in your JavaScript programs.

For example: a === 4

This code tests to see whether the value of the variable a is equal to the number 4. If it is, the result of this comparison is the boolean value true. If a is not equal to 4, the result of the comparison is false. Boolean values are commonly used in JavaScript control structures. For example, the if/else statement in JavaScript performs one action if a boolean value is true and another action if the value is false. You usually combine a comparison that creates a boolean value directly with a statement that uses it. The result looks like this: if (a === 4) { b = b + 1; } else { a = a + 1; } This code checks whether a equals 4. If so, it adds 1 to b; otherwise, it adds 1 to a. As we'll discuss in §3.9, any JavaScript value can be converted to a boolean value. The following values convert to, and therefore work like, false: undefined null 0 -0 NaN "" // the empty string All other values, including all objects (and arrays) convert to, and work like, true. false, and the six values that convert to it, are sometimes called falsy values, and all other values are called truthy. Any time JavaScript expects a boolean value, a falsy value works like false and a truthy value works like true. As an example, suppose that the variable o either holds an object or the value null. You can test explicitly to see if o is non-null with an if statement like this: if (o !== null) ... The not-equal operator !== compares o to null and evaluates to either true or false. But you can omit the comparison and instead rely on the fact that null is falsy and objects are truthy: if (o) ... In the first case, the body of the if will be executed only if o is not null. The second case is less strict: it will execute the body of the if only if o is not false or any falsy value (such as null or undefined). Which if statement is appropriate for

your pro- gram really depends on what values you expect to be assigned to o. If you need to distinguish null from 0 and "", then you should use an explicit comparison

Boolean values have a toString() method that you can use to convert them to the strings “true” or “false”, but they do not have any other useful methods. Despite the trivial API, there are three important boolean operators. The && operator performs the Boolean AND operation. It evaluates to a truthy value if and only if both of its operands are truthy; it evaluates to a falsy value otherwise. The || operator is the Boolean OR operation: it evaluates to a truthy value if either one (or both) of its operands is truthy and evaluates to a falsy value if both operands are falsy. Finally, the unary ! operator performs the Boolean NOT operation: it evalu- ates to true if its operand is falsy and evaluates to false if its operand is truthy.

For example: if ((x === 0 && y === 0) || !(z === 0)) { // x and y are both zero or z is non-zero }

3.5 null and undefined

undefined

It means a variable declared, but no value has been assigned a value.

For example,

```
var demo;
alert(demo); //shows undefined
alert(typeof demo); //shows undefined
```

null

Whereas, null in JavaScript is an assignment value. You can assign it to a variable.

For example,

```
var demo = null;
alert(demo); //shows null
alert(typeof demo); //shows object
```

(Triple equals (===) are strict equality comparison operator, which returns false for different types and different content.)

null is a language keyword that evaluates to a special value that is usually used to indicate the absence of a value. Using the typeof operator on null returns the string “object”, indicating that null can be thought of as a special object value that indicates “no object”. In practice, however, null is typically regarded as the sole member of its own type, and it can be used to indicate “no value” for numbers and

strings as well as objects. Most programming languages have an equivalent to JavaScript's null: you may be familiar with it as NULL, nil, or None. JavaScript also has a second value that indicates absence of value. The undefined value represents a deeper kind of absence. It is the value of variables that have not been initialized and the value you get when you query the value of an object property or array element that does not exist. The undefined value is also the return value of functions that do not explicitly return a value and the value of function parameters for which no argument is passed. undefined is a predefined global constant (not a language keyword like null, though this is not an important distinction in practice) that is initialized to the undefined value. If you apply the typeof operator to the unde fined value, it returns "undefined", indicating that this value is the sole member of a special type. Despite these differences, null and undefined both indicate an absence of value and can often be used interchangeably. The equality operator == considers them to be equal. (Use the strict equality operator === to distinguish them.) Both are falsy values: they behave like false when a boolean value is required. Neither null nor undefined

have any properties or methods. In fact, using. or [] to access a property or method of these values causes a TypeError. I consider undefined to represent a system-level, unexpected, or error-like absence of value and null to represent a program-level, normal, or expected absence of value. I avoid using null and undefined when I can, but if I need to assign one of these val- ues to a variable or property or pass or return one of these values to or from a func- tion, I usually use null. Some programmers strive to avoid null entirely and use undefined in its place wherever they can

3.6 Symbols

The JavaScript ES6 introduced a new primitive data type called Symbol. Symbols are immutable (cannot be changed) and are unique. For example,

```
// two symbols with the same description

const value1 = Symbol('hello');
const value2 = Symbol('hello');

console.log(value1 === value2); // false
```

Symbols were introduced in ES6 to serve as non-string property names. To under- stand Symbols, you need to know that JavaScript's fundamental Object type is an unordered collection of properties, where each property has a name and a value. Property names are typically (and until ES6, were exclusively) strings. But in ES6 and later, Symbols can also serve this purpose:

let strname = "string name"; // A string to use as a property name

let symname = Symbol("propname"); // A Symbol to use as a property name

3.7 The Global Object

The preceding sections have explained JavaScript's primitive types and values. Object types—objects, arrays, and functions—are covered in chapters of their own later in this book. But there is one very important object value that we must cover now. The global object is a regular JavaScript object that serves a very important purpose: the properties of this object are the globally defined identifiers that are available to a Java- Script program. When the JavaScript interpreter starts (or whenever a web browser loads a new page), it creates a new global object and gives it an initial set of properties that define: • Global constants like undefined, Infinity, and NaN • Global functions like isNaN(), parseInt() (§3.9.2), and eval() (§4.12)

• Constructor functions like Date(), RegExp(), String(), Object(), and Array() (§3.9.2) • Global objects like Math and JSON (§6.8) The initial properties of the global object are not reserved words, but they deserve to be treated as if they are. This chapter has already described some of these global properties. Most of the others will be covered elsewhere in this book. In Node, the global object has a property named global whose value is the global object itself, so you can always refer to the global object by the name global in Node programs. In web browsers, the Window object serves as the global object for all JavaScript code contained in the browser window it represents. This global Window object has a selfreferential window property that can be used to refer to the global object. The Win- dow object defines the core global properties, but it also defines quite a few other globals that are specific to web browsers and client-side JavaScript. Web worker threads (§15.13) have a different global object than the Window with which they are associated. Code in a worker can refer to its global object as self. ES2020 finally defines globalThis as the standard way to refer to the global object in any context. As of early 2020, this feature has been implemented by all modern browsers and by Node.

3.8 Immutable Primitive Values and Mutable Object References

Now, these data types are broadly classified into 2 types:

Primitive:- (String,Boolean,Number,BigInt,Null,Undefined,Symbol)

Non-Primitive:- Object (array, functions) also called object references.

The fundamental difference between primitives and non-primitive is that primitives are immutable and non-primitive are mutable.

Mutable values are those which can be modified after creation.

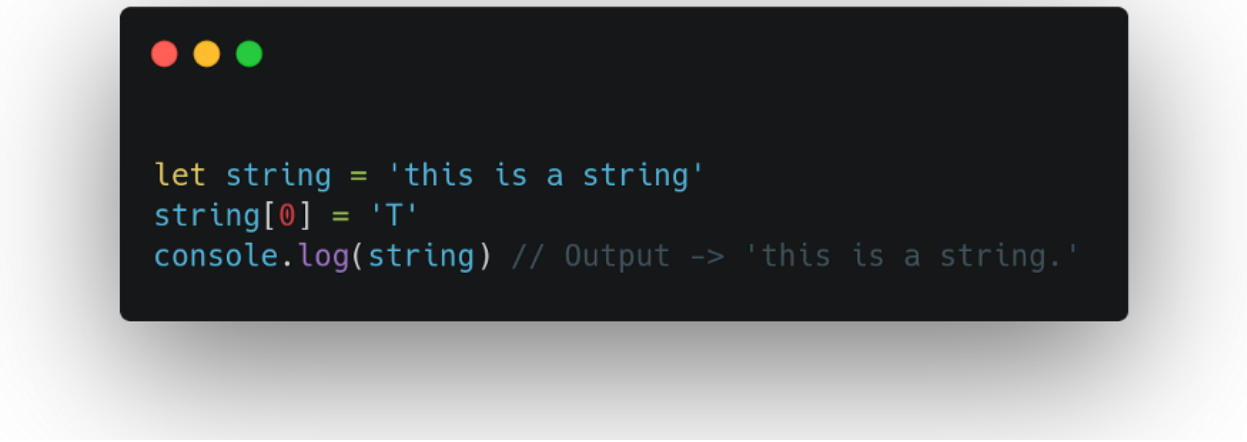
Immutable values are those which cannot be modified after creation.

Primitives are known as being immutable data types because there is no way to change a primitive value once it gets created.

Example1:-

primitive is immutable (cannot be directly altered)

It’s important to note in the above example that a variable that stored primitive value can be reassigned to a new value as shown in the above example but the existing value can’t be changed as shown below:-



primitive values are immutable

Non-Primitives are known as mutable data types because we can change the value after creation.

```
let arr = [1,2,3,4,5];
arr[1] = 7;

console.log(arr); // [1,7,3,4,5]
```

non-primitive values are mutable

As you can see in the above example we can change the array after creation.

There is a fundamental difference in JavaScript between primitive values (undefined, null, booleans, numbers, and strings) and objects (including arrays and functions). Primitives are immutable: there is no way to change (or “mutate”) a primitive value. This is obvious for numbers and booleans—it doesn’t even make sense to change the value of a number. It is not so obvious for strings, however. Since strings are like arrays of characters, you might expect to be able to alter the character at any specified index. In fact, JavaScript does not allow this, and all string methods that appear to return a modified string are, in fact, returning a new string value. For example: let s = "hello"; // Start with some lowercase text s.toUpperCase(); // Returns "HELLO", but doesn't alter s s // => "hello": the original string has not changed Primitives are also compared by value: two values are the same only if they have the same value. This sounds circular for numbers, booleans, null, and undefined: there is no other way that they could be compared. Again, however, it is not so obvious for strings. If two distinct string values are compared, JavaScript treats them as equal if, and only if, they have the same length and if the character at each index is the same.

Objects are different than primitives. First, they are mutable—their values can change: let o = { x: 1 }; // Start with an object o.x = 2; // Mutate it by changing the value of a property o.y = 3; // Mutate it again by adding a new property let a = [1,2,3]; // Arrays are also mutable a[0] = 0; // Change the value of an array element a[3] = 4; // Add a new array element Objects are not compared by value: two distinct objects are not equal even if they have the same properties and values. And two distinct arrays are not

equal even if they have the same elements in the same order: let o = {x: 1}, p = {x: 1}; // Two objects with the same properties o === p // => false: distinct objects are never equal let a = [], b = []; // Two distinct, empty arrays a === b // => false: distinct arrays are never equal Objects are sometimes called reference types to distinguish them from JavaScript's primitive types. Using this terminology, object values are references, and we say that objects are compared by reference: two object values are the same if and only if they refer to the same underlying object. let a = []; // The variable a refers to an empty array. let b = a; // Now b refers to the same array. b[0] = 1; // Mutate the array referred to by variable b. a[0] // => 1: the change is also visible through variable a. a === b // => true: a and b refer to the same object, so they are equal. As you can see from this code, assigning an object (or array) to a variable simply assigns the reference: it does not create a new copy of the object. If you want to make a new copy of an object or array, you must explicitly copy the properties of the object or the elements of the array. This example demonstrates using a for loop (§5.4.3): let a = ["a","b","c"]; // An array we want to copy let b = []; // A distinct array we'll copy into for(let i = 0; i < a.length; i++) { // For each index of a[] b[i] = a[i]; // Copy an element of a into b } let c = Array.from(b); // In ES6, copy arrays with Array.from() Similarly, if we want to compare two distinct objects or arrays, we must compare their properties or elements. This code defines a function to compare two arrays: function equalArrays(a, b) { if (a === b) return true; // Identical arrays are equal if (a.length !== b.length) return false; // Different-size arrays not equal for(let i = 0; i < a.length; i++) { // Loop through all elements if (a[i] !== b[i]) return false; // If any differ, arrays not equal }