3.1 Storing Values in Variables

R E A D Y

CODE DIFFERENTLY

2023

Why do we write programs?

To declare a variable in JavaScript, you can use the var keyword or the let keyword, followed by the variable name

You should use the let keyword if the corresponding JavaScript(JS) runtime environment supports it

Code

```
var firstName;

var lastName;
```

Code

```
let firstName;

let lastName;
```

Strict Mode

Strict Mode

mode ensures that potentially error-prone features cannot be used within a script

- Eliminates some JavaScript silent errors by changing them to throw errors.
- Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
- Prohibits some syntax likely to be defined in future versions of ECMAScript.

You can activate strict mode either for the entire script or specifically for individual functions. To use for the entire script, include the statement 'use strict' at the

Code

```
'use strict';

let firstName;

let lastName;
```

Variable Assignment | Variable Initialization

Code

```
let firstName;

let lastName;

firstName = "John";

lastName = "Doe";
```

Code

```
let firstName = "John";

let lastName = "Doe";
```

Code

```
let firstName = "John", lastName = "Doe";
```

After declaring a variable, you can **assign** an actual value to it by using the equals sign (=), you assign a concrete value to a variable. The equals sign is the **assignment operator**

Variable declaration and variable initialization can be combined into one step. JavaScript also allows developers to declare and initialize several variables within a single statement

Variable Assignment | Variable Initialization

Keywords in JavaScript are a set of reserved words that cannot be used as names of functions, labels, or variables as they are already a part of the syntax of JavaScript.

All keywords are reserved for current or later use within JavaScript—you still must not use any of these keywords as variables

JavaScript Keywords							
alert	class	delete	finally	import	new	screen	typeof
async	close	do	focus	in	open	static	var
await	closed	documen	for	instanceof	package	super	void
blur	const	t else	frames	interface	private	switch	while
break	continue	enum	function	let	protected	this	window
case	debugger	export	if	location	public	throw	with
catch	default	extends	implements	navigator	return	try	yield

Allowed Characters

Regardless of keywords and predefined variable names, you may use only certain characters within variable names, including letters and numbers, and you can't start a variable name with a number.

JavaScript

```javascript
const 2ndName = 'James';  // invalid because it starts with a number

const first%Name = 'John'; // invalid because it contains special characters

const first-name = 'John'; // invalid because it contains a hyphen

const first_name = 'John'; // valid

const _firstName = 'John'; // valid

const $firstName = 'John'; // valid
```

Case Sensitivity

Variable names are case-sensitive, for example, name, Name, and nAme each represent different variables

JavaScript

```
const name = 'John';  // This is a different variable ...

const Name = 'Bob';  // ... from this variable ...

const nAme = 'Pete';  // ... and this variable.
```

CamelCase Spelling

A generally accepted notation, for example, is the lowerCamelCase notation, where the variable name starts with a lowercase letter and then, if the name is composed of several words, each of the other words starts with an uppercase letter

JavaScript

```javascript
const defaultValue = 2345;

const firstName = 'John';

const lastName = 'Doe';

const isAdmin = true;

const userIsNotLoggedIn = true;
```

JavaScript

```javascript
class Rectangle {}
```

There is also the UpperCamelCase notation used for class names, the only difference is that the first letter is uppercase instead of lowercase

Meaningful Names

Choose variables names that are as meaningful as possible so that you can tell the purpose of the variable from the name alone

JavaScript

```
// not very meaningful variable names

const fn = 'John ';

const ln = 'Doe ';


// meaningful variable names

const firstName = 'John ';

const lastName = 'Doe ';
```

Defining Constants

Constants are variables that are not variable but constant: once assigned, the value of a constant remains the same
Use the const keyword to define a constant like you would a variable

Code

```
const MAXIMUM = 5000;


// potential runtime error
MAXIMUM = 4711;


console.log(MAXIMUM);
```

Output

```
> 5000
```

3.2 Using the Different Data Types

R E A D Y

CODE DIFFERENTLY

2023

Using the Different Data Types

JavaScript distinguishes among six different data types:

Primitive data types:	Special data types:
String	Null
Numbers	Undefined
Boolean	Object

Loose Typing

JavaScript does not require you to specify a data type in the variable declaration

In JavaScript, data types are determined dynamically at runtime based on the value assigned to the variable

When the data type is not explicitly specified it is referred to as loose typing

Other languages, like Java, use strong typing or strict typing; the data type must be explicitly defined in the variable declaration

JavaScript

```
const num = 50

const bool = true

const str = 'Hello World'
```

Java

```
int num = 50;

boolean bool = true;

String str = "Hello World";
```

Numbers

Numbers

Numbers are represented by their numeric value. Negative values can also be defined by simply prefixing the respective number with a minus sign

Code

```
const integer = 5;   // integer

const decimal = 0.5;   // decimal

const negInteger = -22;  // negative integer

const negDecimal = -0.9;  // negative decimal

const separator = 12_300;  // separator

const separator2 = 1_000_000; // separator


console.log(integer);

console.log(decimal);

console.log(negInteger);

console.log(negDecimal);

console.log(separator);

console.log(separator2);
```

Output

```
> 5

> 0.5

> -22

> -0.9

> 12300

> 1000000
```

Number Systems

In addition to the decimal notation (base 10), integers can be defined in JavaScript using different number systems

To use each system, use the syntax specific to the notation

Notation	Base	Example	Syntax
Decimal	10	85	Numbers 0-9
Binary	2	0b01010101	prefix 0b followed by a sequence of zeros and ones
Octal	8	0125	start with a 0 followed by a sequence of numbers between 0 and 7
Hexadecimal	16	0x55	start with the string 0x, followed by a sequence of hexadecimal values: the digits 0 to 9 and the letters A to F

Range of Numbers

The value range of numbers is limited, you cannot define infinitely large or infinitely small numbers

In most runtime environments, the range will be 5e-324 to 1.7976931348623157e+308

The maximum and minimum numbers are stored as properties in the Number object, as MIN_VALUE and MAX_VALUE

If a calculation requires an actual result outside this value range, the Infinity value can be used. The Number object also holds properties representing infinity: -Infinity and Infinity

Code

```
console.log(Number.MIN_VALUE);
console.log(Number.MAX_VALUE);

console.log(-Infinity);
console.log(Infinity);
```

Output

```
> 5e-324

> 1.7976931348623157e+308

> -Infinity

> Infinity
```

Strings

Meaningful Names

Code

```
const str = 'Hello World!';
```

Strings are sequences of characters—for example, letters, digits, special characters, and control characters

The beginning and the end of a string are defined by quotation marks, in JavaScript you can use both single and double quotation marks

Best Practices:

- Define strings using single quotes rather than double quotes, so you avoid having to escape the double quotes
- Do not to mix the types of quotes within a program, decide on a type and stick to it consistently

Escaping Characters within Strings

Constants are variables that are not variable but constant: once assigned, the value of a constant remains the same
Use the const keyword to define a constant like you would a variable

Code

```
const message1 = 'Your name is "John Doe"';
const message2 = "Your name is 'John Doe'";
const message3 = 'Your name is \'John Doe\'';
const message4 = "Your name is \"John Doe\"";

console.log(message1);
console.log(message2);
console.log(message3);
console.log(message4);
```

Output

```
> Output: Your name is "John Doe"

> Output: Your name is 'John Doe'

> Output: Your name is 'John Doe'

> Output: Your name is "John Doe"
```

Control Characters

Control characters, which are characters that cannot be represented as such—for example, line breaks, indentations, and so on can be represented by using the backslack (\) and special characters

Characters	Meaning
\n	New line
\t	Tab character/indentation
\b	Backspace
\r	Carriage return
\f	Form feed

String Concatenation

In everyday work with strings, you'll often want to insert calculated values or values stored in variables at certain positions within a string. As a rule, string concatenation is then used to assemble the individual parts into a string

Code

```
const firstName = 'John';
const lastName = 'Doe';
const age = 44;
const message = 'My name is ' + firstName + ' ' +
lastName + ', I am ' + age + ' years old.';
console.log(message);
```

Output

```
> My name is John Doe, I am 44 years old.
```

Using Template Strings

Template strings are like an extended form of normal strings, they are defined by ` characters (called backticks)

Within template strings, you can define placeholders using the ${} notation, which populates the two placeholders with the values of the corresponding variables

Code

```
const firstName = 'John';
const lastName = 'Doe';
const age = 44;
const message = `My name is ${firstName}
${lastName}, I am ${age} years old.`;
console.log(message);
```

Output

```
> My name is John Doe, I am 44 years old.
```

Evaluating Expressions within Strings

You can use any other expressions inside the curly braces pf a template string

Code

```
const name = 'John Doe';
const age = 44;

function getName() {
  return name;

const message = `My name is ${getName()}, I am
twice the age of ${age/2}.`;
console.log(message);
```

Output

```
> My name is John Doe, I am twice the age of

22.
```

Defining Multiline Strings

If you are dealing with long strings, in practice you will probably often split these strings into several parts and then distribute them over several lines, line breaks are acceptable within template strings

Line breaks that are to be included in the string output must be coded with \n

Code

```
const message = `Dear Mr. Doe, \n\nWe are happy to return the requested documents to you for review. \n\nYours sincerely, \nMrs. Smith, \nSample Company`;

console.log(message);
```

Output

```
> Dear Mr. Doe,

We are happy to return the requested documents to you for review.

Yours sincerely,
Mrs. Smith,
Sample Company
```

Boolean

Boolean Values

Code

```
const isLoggedIn = true;

const isAdmin = false;
```

Boolean values or Booleans can only take one of two values: true and false

Booleans are usually a good choice when a variable can take one of two values

Arrays

Arrays

Code

```
const fruits = ['apple',
'cherry', 'peach'];
const veggies = [
  'corn',
  'pepper',
  'beet'
];
```

Arrays are lists, they can contain not only one but several values

The most common and easiest way to create an array in JavaScript is the array literal notation, use the two square brackets ([]) to define the beginning and the end of the array, with individual entries between these two brackets, separated by commas

Loose Typing in Arrays

In JavaScript, arrays can also contain entries of different types, one array can contain numbers as well as strings or Boolean values, objects, and even other arrays

Code

```
const values = [
  'Sally Doe',
  true,
  542_234,
  ['dog','cat','bird']
];
```

This array contains a string, boolean value, number, and another array

Internal Structure of Arrays

Arrays have an index-based structure, every element in an array is stored at a particular index

The counting of these indexes starts at 0, the second element is at index 1, and so on until the last element is at index n-1, where n is the length of the array

Use the indexes to specifically access the individual elements of an array, simply write the index of the element you want to access in square brackets after the name of the respective array

Code

```
const fruits = ['apple', 'cherry', 'peach'];

const fruit1 = fruits[0];
const fruit2 = fruits[1];
const fruit3 = fruits[2];

console.log(fruit1);
console.log(fruit2);
console.log(fruit3);
```

Array Structure

Index	Value
0	apple
1	cherry
2	peach

Output

```
> apple
> cherry
> peach
```

Multidimensional Arrays

It’s also possible to add other arrays to an array, this enables you to create multidimensional arrays, i.e. arrays of arrays

To access an element, access each dimension of the array separately, with square brackets

Code

```
const food = [
  ['apple', 'cherry', 'peach'],
  ['corn', 'pepper', 'beet']
];

const food1 = food[0][0];
const food2 = food[0][2];
const food3 = food[1][1];

console.log(food1);
console.log(food2);
console.log(food3);
```

Array Structure

Index	Value
0	apple
1	cherry

Array Structure

Index	Value
0	apple
1	cherry
2	peach

Index	Value
0	corn
1	pepper
2	beat

Output

```
> apple
> cherry
> peach
```

Objects

Objects

It is better—and common practice in object-oriented programming—to combine related variables and functions in objects

In the context of objects, variables are called properties (object properties) or attributes, and functions are referred to as methods (object methods)

Properties provide information about the object they are contained in, such as a person's name or age, the items in a shopping cart, or the number of links on a web page

Methods represent certain tasks related to the object in which they are defined, such as adjusting a person's age or adding an item to a shopping cart

Creating Objects

The easiest way to create objects in JavaScript is to use object literal notation, define the object using braces and list the object properties and object methods within these braces, separated by commas

Objects store data in key-value pairs, when defining an object the key and value are separated from each other by a colon

Code

```
const dog = {
  name: 'Pluto',
  age: 85,
  breed: 'Cartoon',
  bark: function() {
   console.log(`${this.name} says "Bark Bark!"`);
  }
}
```

Object Structure

Dog	
name	Pluto
age	85
breed	Cartoon
bark	function()

Special Types

Special Data Types

There are two more special data types in JavaScript that are often confused in practice and/or used for the same thing

If you declare a variable but do not initialize it yet (by assigning a value to it), it has the value undefined

The null data type represents an empty object

The undefined value is not meant to be manually assigned to a variable, but the null value is

Symbols

Symbols are another kind of primitive data type, they enable you to define unique and immutable values

Symbols can be created using the Symbol() function, optionally passing a symbol description as a parameter

Code

```
const symbol1 = Symbol();
const symbol2 = Symbol('exampleDescription');
const symbol3 = Symbol();
const symbol4 = Symbol('exampleDescription');

console.log(symbol1);
console.log(symbol2);
console.log(symbol3);
console.log(symbol4);
console.log(symbol1 === symbol3);
console.log(symbol2 === symbol4)
```

Output

```
> Symbol()

> Symbol(exampleDescription)

> Symbol()

> Symbol(exampleDescription)

> false

> false
```

3.3 Deploying the Different Operators

R E A D Y

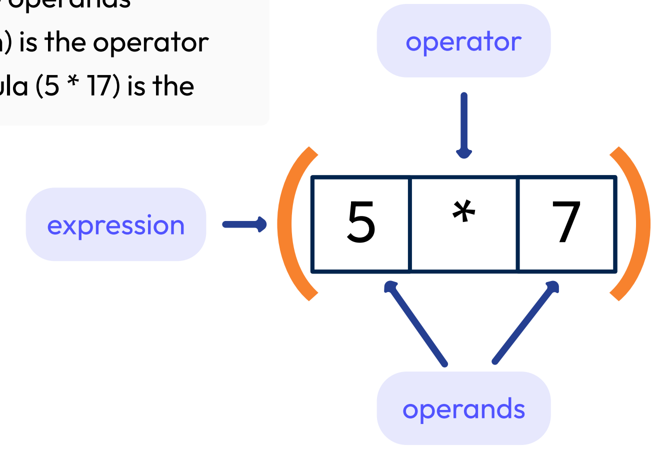
2023

Deploying the Different Operators

Similar to a mathematical formula, a code expression consists of operands and operators

Here is the example (5 * 17)

- The 5 and 17 are the operands
- The * (multiplication) is the operator
- The complete formula (5 * 17) is the expression



Statements vs Expressions

The terms statement and expression are often confused. In 3.2, we worked with variable assignments, which were statements

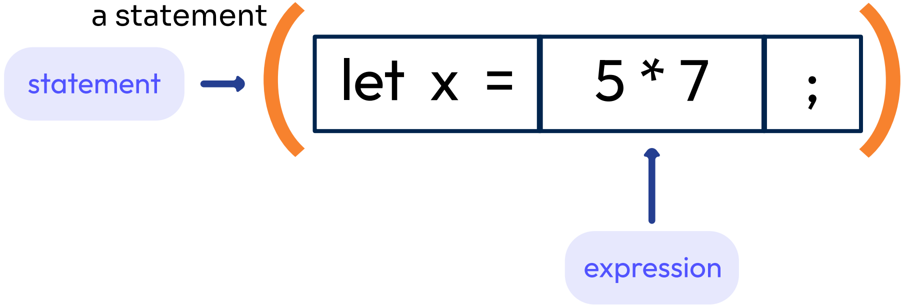
Statements are code that is to be executed; however, expressions have a result that is returned

- Each statement is terminated with a semicolon, this indicates to the interpreter that a statement is complete

- Expressions are combinations of operators and operands that together yield a value and can be part of a statement

Code

```
let x = 5 * 17;
```



Arithmetic Operators

JavaScript provides a set of arithmetic operators—that is, operators for working and calculating with numbers

Notation	Operator	Example	Result of x
Addition	+	let x = 20 + 20	40
Subtraction	-	let x = 20 - 5	15
Multiplication	*	let x = 20 * 5	100
Division	/	let x = 100 / 5	20
Modulo	%	let x = 20 % 6	2
Increment	++	let x = 5; x++;	6
Decrement	--	let x = 6; x--;	5
Power calculation	**	let x = 5**5;	3125

Shorthand Operators

As an alternative to this long notation, you can also use a short notation for the addition, subtraction, multiplication, division, and remainder operators, which combines both steps: the operation and the assignment of the result to the corresponding variable

Shorthand Opertator	Short Form	Long Form
+=	result += 11	result = result + 11
-=	result -= 11	result = result - 11
*=	result *= 11	result = result * 11
/=	result /= 11	result = result / 11
%=	result %= 11	result = result % 11
**=	result **= 11	result = result ** 11

Operators for Strings

In JavaScript, the operator for joining two strings is the (+) operator, the joining itself is also called concatenation

Code

```
let firstName = 'John';
let lastName = 'Doe';

let fullName = firstName + ' ' + lastName;

console.log(firstName);
console.log(lastName);
console.log(fullName);
```

Output

```
> John

> Doe

> John Doe
```

Logical Operators

For working with Boolean values, there are various operators available in programming, which are referred to as logical operators

The AND operator and OR operator are binary operators, they expect two operands

The negation operator is a unary operator, it expects only a single operand

The result of all Boolean operations is a Boolean value, all expressions will return true or false

Array Structure	
&&	Logical AND
||	Logical OR
!	Negation

Code

```
const isLoggedIn = true;
const isAdmin = false;

console.log(isLoggedIn && isAdmin);
console.log(isLoggedIn || isAdmin);
console.log(!isLoggedIn);
```

Output

```
> false
> true
> false
```

Logical Combinations

This shows the results of the AND, OR, and negation operators

Array Structure			
Operand 1	Operand 2	Expression	Result
true	true	true && true	true
true	false	true && false	false
false	true	false && true	false
false	false	false && false	false

Array Structure			
Operand 1	Operand 2	Expression	Result
true	true	true && true	true
true	false	true && false	false
false	true	false && true	false
false	false	false && false	false

Array Structure		
Operand 1	Expression	Result
true	!true	false
false	!false	true

Code

```
const isLoggedIn = true;
const isAdmin = false;
console.log(isLoggedIn && isAdmin);
console.log(isLoggedIn || isAdmin);
console.log(!isLoggedIn);
```

Output

```
> false
> true
> false
```

Logical Operators with Non-Boolean Values

Boolean operators can also be used with operands that are not Boolean values, such as numbers, strings, or objects

In the case of a logical AND operation:

- If the first operand evaluates to false, the first operand is returned
- In all other cases, the second operand is returned

In the case of logical OR operation:

- If the first operand evaluates to true, the first operand is returned
- In all other cases, the second operand is returned

In the case of negation:

- An empty string, the number 0, and the special values null, NaN, and undefined all return true
- A nonempty string, all numbers except 0 (including the special Infinity value), and Objects all return false

Array Structure

Operand	Result
Empty string	true
Nonempty string	false
Number 0	true
Number not equal to 0 (including Infinity)	false
Object	false
null	true
NaN	true
undefined	true

Nullish Coalescing Operator

The nullish coalescing operator, consists of two consecutive question marks

The ?? operator returns the value of the right operand only if the left operand is null or undefined

Code

```
let valueA;
let valueB = "My defined value";

const result = valueA ?? valueB;
console.log(result);
```

Output

```
> My defined value
```

Bitwise Operators

Bitwise operations enable you to work with single bits of values

Operator	Result
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise NOT
<<	Bitwise left shift
>>	Bitwise right shift
>>>	Bitwaise unsigned right shift

Output

```
let BYTE_A = 0b00000011; // Binary 00000011, decimal 3
let BYTE_B = 0b01000001; // Binary 01000001, decimal 65

// bitwise left shift
BYTE_A = BYTE_A << 1;  // Binary 00000110, decimal 6

// bitwise right shift
BYTE_A = BYTE_A >> 1;  // Binary 00000011, decimal 3

// bitwise AND
let BYTE_C = BYTE_A & BYTE_B; // Binary 00000001, decimal 1

// bitwise OR
let BYTE_D = BYTE_A | BYTE_B; // Binary 01000011, decimal 67

// bitwise exclusive OR
let BYTE_E = BYTE_A ^ BYTE_B; // Binary 01000010, decimal 66
```

Operators for Comparing Values

Comparing two values is certainly one of the most frequently performed tasks in programming, be it comparing strings, numbers, or Boolean values, these comparison operators return a true or false value

Operator	Meaning
==	equal
!=	Unequal to
===	Strictly equal to
!==	Strictly unequal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Code

```
console.log(false == 0);
console.log(false == 1);
console.log(true == 1);
console.log(true == 0);
console.log("4711" == 4711);
console.log(false != 0);
console.log(false != 1);
console.log(true != 1);
console.log(true != 0);
console.log("4711" != 4711);
```

Output

```
> true
> false
> true
> false
> true
> false
> true
> false
> true
> false
```

Implicit Type Conversion

The automatic conversion of a value into another data type is also called implicit type conversion

The interpreter tries to convert the second operand into the data type of the first operand

Code

```
console.log(false == 0);
console.log(true == 1);
console.log("4711" == 4711);
```

Output

```
> true

> true

> true
```

Nonstrict Comparison versus Strict Comparison

The operators == and != perform a nonstrict comparison of two values, whereas the operators === and !== perform a strict comparison of two values

The == and != operators only compare the two values

The === and !== operators compare values and the data types of the corresponding values

Code

```
console.log(false === 0);
console.log(false === 1);
console.log(true === 1);
console.log(true === 0);
console.log("4711" === 4711);
console.log(false !== 0);
console.log(false !== 1);
console.log(true !== 1);
console.log(true !== 0);
console.log("4711" !== 4711);
```

Output

```
> false
> false
> false
> false
> false
> true
> true
> true
> true
> true
```

The Optional Chaining Operator

In ES2020, the optional chaining operator was included, this operator is placed directly after a property when it is accessed, which ensures that the next hierarchy level is only accessed if the corresponding property exists

Code

```
const john = {
  firstName: 'John',
  lastName: 'Doe',
  contact: {
    email: 'john.doe@email.com'
  }
};
const sally = {};

console.log(john.contact?.email);
console.log(sally.contact?.email);
```

Output

```
> john.doe@email.com
> undefined
```

The Logical Assignment Operators

The logical assignment operators are operators that combine logical operators and assignment expressions, making conditional value assignment easier for variables and object properties

Code

```
let a1 = 5;
let a2 = null;
let a3 = false;
a1 ||= 7;
a2 ||= 7;
a3 ||= 7;
console.log(`a1: ${a1}`);
console.log(`a2: ${a2}`);
console.log(`a3: ${a3}`);
```

Output

```
> a1: 5
> a2: 7
> a3: 7
```

Operator	Name
||=	logical OR assignment operator
&&=	logical AND assignment operator
??	logical nullish assignment operator

Ternary operator

Ternary operator that, depending on a condition (first operand), returns one of two values (defined by the second and third operands)

<condition> ? <value1> : <value2>

Code

```
const john = {
 firstName: 'John',
 lastName: 'Doe',
 contact: {
 email: 'john.doe@email.com'
 }
};
const sally = {};

console.log(john.contact?.email);
console.log(sally.contact?.email);
```

Output

```
> B
> C
```

Special Operations

The logical assignment operators are operators that combine logical operators and assignment expressions, making conditional value assignment easier for variables and object properties

Operator	Operation
delete	Deleting objects, object properties, or elements within an array
<property> in <object>	Checks whether an object has a property
<object> instanceof <type>	Binary operator that checks whether an object is of a specific type
typeof <Operand>	Determines the data type of the operand. The operand can be an object, a string, a variable, or a keyword like true or false

Code

```
let obj = { firstName: 'John', lastName: 'Doe' };
delete obj.firstName;
console.log(obj);

console.log(lastName in obj);

console.log(typeof obj);
```

Output

```
> { lastName: 'Doe' }

> true

> object
```