New Horizons Computer Learning Centers

UPGRADING PEOPLE EVERY DAY

# JavaScript Introduction

*Svetlomir Miloslavov*

# Before We Start

- NotePad++: https://notepad-plus-plus.org/downloads/
- Chrome: https://www.google.com/chrome/
- Visual Studio Code: https://code.visualstudio.com/download
- XAMPP: https://www.apachefriends.org/index.html

# What is JavaScript?

JavaScript is the world's most popular programming language.

JavaScript is the programming language of the Web.

JavaScript is easy to learn.

# What is JavaScript?

- How do I get JavaScript?
- Where can I download JavaScript?
- Is JavaScript Free?

**You don't have to get or download JavaScript.**

**JavaScript is already running in your browser on your computer, on your tablet, and on your smart-phone.**

**JavaScript is free to use for everyone.**

# What is JavaScript?

- **JavaScript Can Change HTML Content**
- **JavaScript Can Change HTML Attribute Values**
- **JavaScript Can Change HTML Styles (CSS)**
- **JavaScript Can Show & Hide HTML Elements**

**Note:** JavaScript and Java are completely different languages, both in concept and design.

JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.

ECMA-262 is the official name of the standard. ECMAScript is the official name of the language.

# JavaScript Getting Started

In HTML, JavaScript code is inserted between `<script>` and `</script>` tags.

```
<script>
document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
```

**Note:** Old JavaScript examples may use a type attribute: <script type="text/javascript">. The type attribute is not required. JavaScript is the default scripting language in HTML.

# JavaScript Getting Started

A JavaScript `function` is a block of JavaScript code, that can be executed when "called" for.

For example, a function can be called when an **event** occurs, like when the user clicks a button.

You can place any number of scripts in an HTML document.

Scripts can be placed in the `<body>`, or in the `<head>` section of an HTML page, or in both.

# JavaScript Getting Started

In this example, a JavaScript `function` is placed in the `<head>` section of an HTML page.

The function is invoked (called) when a button is clicked:
https://www.w3schools.com/js/tryit.asp?filename=tryjs_whereto_head

**NOTE**: You can place the scripts in the body section as well, they will still work!

# JavaScript Getting Started

Scripts can also be placed in external files.

External scripts are practical when the same code is used in many different web pages.

JavaScript files have the file extension **.js**.

To use an external script, put the name of the script file in the `src` (source) attribute of a `<script>` tag:

```
<script src="myScript.js"></script>
```

**NOTE:** External scripts cannot contain `<script>` tags.

# JavaScript Getting Started

Placing scripts in external files has some advantages:

- It separates HTML and code
- It makes HTML and JavaScript easier to read and maintain
- Cached JavaScript files can speed up page loads

External scripts can be referenced with a full URL or with a path relative to the current web page.

```
<script src="myScript1.js"></script>
<script src="myScript2.js"></script>

<script src="https://www.w3schools.com/js/myScript1.js"></script>
```

# JavaScript Output

JavaScript can "display" data in different ways:

- Writing into an HTML element, using `innerHTML`.
- Writing into the HTML output using `document.write()`.
- Writing into an alert box, using `window.alert()`.
- Writing into the browser console, using `console.log()`.

# JavaScript Output

To access an HTML element, JavaScript can use the `document.getElementById(id)` method.

The `id` attribute defines the HTML element.

The `innerHTML` property defines the HTML content:
https://www.w3schools.com/js/tryit.asp?filename=tryjs_output_dom

**Note:** Changing the innerHTML property of an HTML element is a common way to display data in HTML.

# JavaScript Output

For testing purposes, it is convenient to use `document.write()`:
https://www.w3schools.com/js/tryit.asp?filename=tryjs_output_write

Using document.write() after an HTML document is loaded, will **delete all existing HTML**:
https://www.w3schools.com/js/tryit.asp?filename=tryjs_output_write_over

**The document.write() method should only be used for testing.**

# JavaScript Output

You can use an alert box to display data:
https://www.w3schools.com/js/tryit.asp?filename=tryjs_output_alert

You can skip the `window` keyword.

In JavaScript, the window object is the global scope object, that means that variables, properties, and methods by default belong to the window object. This also means that specifying the `window` keyword is optional.

# JavaScript Output

For debugging purposes, you can call the `console.log()` method in the browser to display data.

https://www.w3schools.com/js/tryit.asp?filename=tryjs_output_console

# JavaScript Output

JavaScript does not have any print object or print methods.

You cannot access output devices from JavaScript.

The only exception is that you can call the `window.print()` method in the browser to print the content of the current window.

https://www.w3schools.com/js/tryit.asp?filename=tryjs_output_print

# JavaScript Statements

A **computer program** is a list of "instructions" to be "executed" by a computer.

In a programming language, these programming instructions are called **statements**.

A **JavaScript program** is a list of programming **statements**.

**Note**: In HTML, JavaScript programs are executed by the web browser.

# JavaScript Statements

JavaScript statements are composed of:

Values, Operators, Expressions, Keywords, and Comments.

This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo":

```
document.getElementById("demo").innerHTML = "Hello Dolly.";
```

The statements are executed, one by one, in the same order as they are written.

JavaScript programs (and JavaScript statements) are often called JavaScript code.

# JavaScript Statements

Semicolons separate JavaScript statements.

Add a semicolon at the end of each executable statement:

```
var a, b, c;      // Declare 3 variables
a = 5;            // Assign the value 5 to a
b = 6;            // Assign the value 6 to b
c = a + b;    // Assign the sum of a and b to c
```

When separated by semicolons, multiple statements on one line are allowed:

```
a = 5; b = 6; c = a + b;
```

On the web, you might see examples without semicolons.
Ending statements with semicolon is not required, but highly recommended.

# JavaScript Statements

JavaScript ignores multiple spaces. You can add white space to your script to make it more readable.

The following lines are equivalent:

```
var person = "Hege";
var person="Hege";
```

A good practice is to put spaces around operators ( = + - * / ):

```
var x = y + z;
```

# JavaScript Statements

For best readability, programmers often like to avoid code lines longer than 80 characters.

If a JavaScript statement does not fit on one line, the best place to break it is after an operator:

```
document.getElementById("demo").innerHTML =
"Hello Dolly!";
```

# JavaScript Statements

JavaScript statements can be grouped together in code blocks, inside curly brackets {...}.

The purpose of code blocks is to define statements to be executed together.

One place you will find statements grouped together in blocks, is in JavaScript functions:

```
function myFunction() {
  document.getElementById("demo1").innerHTML = "Hello Dolly!";
  document.getElementById("demo2").innerHTML = "How are you?";
}
```

# JavaScript Statements

JavaScript statements often start with a **keyword** to identify the JavaScript action to be performed.

Visit our Reserved Words reference to view a full list of JavaScript keywords.

https://www.w3schools.com/js/js_reserved.asp

# JavaScript Syntax

JavaScript syntax is the set of rules, how JavaScript programs are constructed:

The JavaScript syntax defines two types of values:

- Fixed values
- Variable values

Fixed values are called **Literals**.

Variable values are called **Variables**.

# JavaScript Syntax

The two most important syntax rules for fixed values are:

1. **Numbers** are written with or without decimals:

```
10.50
```

```
1001
```

2. **Strings** are text, written within double or single quotes:

```
"John Doe"
```

```
'John Doe'
```

# JavaScript Syntax

In a programming language, **variables** are used to **store** data values.

JavaScript uses the `var` keyword to **declare** variables.

An **equal sign** is used to **assign values** to variables.

In this example, x is defined as a variable. Then, x is assigned (given) the value 6:

```
var x;

x = 6;
```

# JavaScript Syntax

JavaScript uses **arithmetic operators** ( + - * / ) to **compute** values:

```
(5 + 6) * 10
```

JavaScript uses an **assignment operator** ( = ) to **assign** values to variables:

```
var x, y;
x = 5;
y = 6;
```

# JavaScript Syntax

Not all JavaScript statements are "executed".

Code after double slashes // or between /* and */ is treated as a **comment**.

Comments are ignored, and will not be executed:

```
var x = 5; // I will be executed

// var x = 6;  I will NOT be executed
```

# JavaScript Syntax

Identifiers are names.

In JavaScript, identifiers are used to name variables (and keywords, and functions, and labels).

The rules for legal names are much the same in most programming languages.

In JavaScript, the first character must be a letter, or an underscore (_), or a dollar sign ($).

Subsequent characters may be letters, digits, underscores, or dollar signs.

Numbers are not allowed as the first character.
This way JavaScript can easily distinguish identifiers from numbers.

# JavaScript Syntax

All JavaScript identifiers are **case sensitive**.

The variables `lastName` and `lastname`, are two different variables:

```
var lastname, lastName;
lastName = "Doe";
lastname = "Peterson";
```

JavaScript does not interpret **VAR** or **Var** as the keyword **var**.

# JavaScript Syntax

Historically, programmers have used different ways of joining multiple words into one variable name:

**Hyphens:**

first-name, last-name, master-card, inter-city.

**Hyphens are not allowed in JavaScript. They are reserved for subtractions.**

**Underscore:**

first_name, last_name, master_card, inter_city.

**Upper Camel Case (Pascal Case):**

FirstName, LastName, MasterCard, InterCity.

**Lower Camel Case:**

JavaScript programmers tend to use camel case that starts with a lowercase letter:

firstName, lastName, masterCard, interCity.

# JavaScript Variables

**JavaScript variables are containers for storing data values.**

In this example, x, y, and z, are variables, declared with the var keyword:

```
var x = 5;
var y = 6;
var z = x + y;
```

# JavaScript Variables

Before 2015, using the var keyword was the only way to declare a JavaScript variable.

The 2015 version of JavaScript (ES6 - ECMAScript 2015) allows the use of the const keyword to define a variable that cannot be reassigned, and the let keyword to define a variable with restricted scope.

The main **difference between let and** var is that scope of a variable defined with let is limited to the block in which it is declared while variable declared with var has the global scope. So we can say that var is rather a keyword which defines a variable globally regardless of block scope.

# JavaScript Variables

All JavaScript **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter
- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are case sensitive (y and Y are different variables)
- Reserved words (like JavaScript keywords) cannot be used as names

JavaScript identifiers are case-sensitive.

# JavaScript Variables

JavaScript variables can hold numbers like 100 and text values like "John Doe".

In programming, text values are called text strings.

JavaScript can handle many types of data, but for now, just think of numbers and strings.

Strings are written inside double or single quotes. Numbers are written without quotes.

If you put a number in quotes, it will be treated as a text string.

# JavaScript Variables

You can declare many variables in one statement.

Start the statement with `var` and separate the variables by **comma**:

```
var person = "John Doe", carName = "Volvo", price = 200;
```

A declaration can span multiple lines:

```
var person = "John Doe",
carName = "Volvo",
price = 200;
```

# JavaScript Variables

In computer programs, variables are often declared without a value. The value can be something that has to be calculated, or something that will be provided later, like user input.

A variable declared without a value will have the value `undefined`.

The variable carName will have the value `undefined` after the execution of this statement:

```
var carName;
```

# JavaScript Variables

If you re-declare a JavaScript variable, it will not lose its value.

The variable `carName` will still have the value "Volvo" after the execution of these statements:

```
var carName = "Volvo";
var carName;
```

# JavaScript Variables

As with algebra, you can do arithmetic with JavaScript variables, using operators like = and +:

```
var x = 5 + 2 + 3;
```

You can also add strings, but strings will be concatenated:

```
var x = "John" + " " + "Doe";
```

If you put a number in quotes, the rest of the numbers will be treated as strings, and concatenated.

```
var x = "5" + 2 + 3;
```

Now what do you think that would do then?

```
var x = 2 + 3 + "5";
```

# JavaScript Variables

Since JavaScript treats a dollar sign as a letter, identifiers containing $ are valid variable names:

```
var $$$ = "Hello World";
var $ = 2;
var $myMoney = 5;
```

Using the dollar sign is not very common in JavaScript, but professional programmers often use it as an alias for the main function in a JavaScript library.

In the JavaScript library jQuery, for instance, the main function `$` is used to select HTML elements. In jQuery `$("p");` means "select all p elements".

# JavaScript Variables

Since JavaScript treats underscore as a letter, identifiers containing _ are valid variable names:

```
var _lastName = "Johnson";
var _x = 2;
var _100 = 5;
```

Using the underscore is not very common in JavaScript, but a convention among professional programmers is to use it as an alias for "private (hidden)" variables.

# JavaScript Variable Exercise

*https://www.w3schools.com/js/exercise_js.asp?filename=exercise_js_variables1*

# JavaScript Operators

Arithmetic operators are used to perform arithmetic on numbers:

| Operator | Description |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| ** | Exponentiation (ES2016) |
| / | Division |
| % | Modulus (Division Remainder) |
| ++ | Increment |
| -- | Decrement |

# JavaScript Operators

Assignment operators assign values to JavaScript variables.

| Operator | Example | Same As |
|---|---|---|
| = | x = y | x = y |
| += | x += y | x = x + y |
| -= | x -= y | x = x - y |
| *= | x *= y | x = x * y |
| /= | x /= y | x = x / y |
| %= | x %= y | x = x % y |
| **= | x **= y | x = x ** y |

# JavaScript Operators

Comparison operators:

| Operator | Description |
|---|---|
| == | equal to |
| === | equal value and equal type |
| != | not equal |
| !== | not equal value or not equal type |
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |
| ? | ternary operator |

# JavaScript Operators

Logical operators:

| Operator | Description |
|---|---|
| && | logical and |
| \|\| | logical or |
| ! | logical not |

Type operators:

| Operator | Description |
|---|---|
| typeof | Returns the type of a variable |
| instanceof | Returns true if an object is an instance of an object type |

# JavaScript Operators

Bitwise operators:

| Operator | Description | Example | Same as | Result | Decimal |
|---|---|---|---|---|---|
| & | AND | 5 & 1 | 0101 & 0001 | 0001 | 1 |
| \| | OR | 5 \| 1 | 0101 \| 0001 | 0101 | 5 |
| ~ | NOT | ~ 5 | ~0101 | 1010 | 10 |
| ^ | XOR | 5 ^ 1 | 0101 ^ 0001 | 0100 | 4 |
| << | Zero fill left shift | 5 << 1 | 0101 << 1 | 1010 | 10 |
| >> | Signed right shift | 5 >> 1 | 0101 >> 1 | 0010 | 2 |
| >>> | Zero fill right shift | 5 >>> 1 | 0101 >>> 1 | 0010 | 2 |

# JavaScript Operators Exercise

*https://www.w3schools.com/js/exercise_js.asp?filename=exercise_js_operators1*

# JavaScript Data Types

JavaScript variables can hold many **data types**: numbers, strings, objects and more:

```
var length = 16;                                    // Number
var lastName = "Johnson";                           // String
var x = {firstName:"John", lastName:"Doe"};         // Object
```

# JavaScript Data Types

In programming, data types is an important concept.

To be able to operate on variables, it is important to know something about the type.

Without data types, a computer cannot safely solve this:

```
var x = 16 + "Volvo";
```

Does it make any sense to add "Volvo" to sixteen? Will it produce an error or will it produce a result?

JavaScript will treat the example above as:

```
var x = "16" + "Volvo";
```

# JavaScript Data Types

JavaScript evaluates expressions from left to right. Different sequences can produce different results:

```
var x = 16 + 4 + "Volvo";  //RESULT: 20Volvo
var x = "Volvo" + 16 + 4; //RESULT: Volvo164
```

In the first example, JavaScript treats 16 and 4 as numbers, until it reaches "Volvo".

In the second example, since the first operand is a string, all operands are treated as strings.

# JavaScript Data Types

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

```
var x;               // Now x is undefined
x = 5;               // Now x is a Number
x = "John";          // Now x is a String
```

# JavaScript Data Types

Booleans can only have two values: `true` or `false`.

```
var x = 5;
var y = 5;
var z = 6;
(x == y)   // Returns true
(x == z)   // Returns false
```

Booleans are often used in conditional testing.

# JavaScript Data Types

JavaScript arrays are written with square brackets.

Array items are separated by commas.

The following code declares (creates) an array called `cars`, containing three items (car names):

```
var cars = ["Saab", "Volvo", "BMW"];
```

Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

# JavaScript Data Types

JavaScript objects are written with curly braces {}.

Object properties are written as name:value pairs, separated by commas.

```
var person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
```

The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.

# JavaScript Data Types

You can use the JavaScript `typeof` operator to find the type of a JavaScript variable.

The `typeof` operator returns the type of a variable or an expression:

```
typeof ""          // Returns "string"
typeof "John"      // Returns "string"
typeof 0           // Returns "number"
typeof 314         // Returns "number"
typeof 3.14        // Returns "number"
typeof (3)         // Returns "number"
```

# JavaScript Data Types

In JavaScript, a variable without a value, has the value `undefined`. The type is also `undefined`.

```
var car;  // Value is undefined, type is undefined
```

In JavaScript `null` is "nothing". It is supposed to be something that doesn't exist.

Unfortunately, in JavaScript, the data type of `null` is an object.

You can consider it a bug in JavaScript that `typeof null` is an object. It should be `null`.

You can empty an object by setting it to `null`:

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = null;  // Now value is null, but type is still an object
```

# JavaScript Data Types

`undefined` and `null` are equal in value but different in type:

```
typeof undefined       // undefined
typeof null            // object

null === undefined     // false
null == undefined      // true
```

# JavaScript Data Types

A primitive data value is a single simple data value with no additional properties and methods.

The `typeof` operator can return one of these primitive types:

- `string`
- `number`
- `boolean`
- `undefined`

```
typeof "John"                 // Returns "string"
typeof 3.14                   // Returns "number"
typeof true                   // Returns "boolean"
typeof false                  // Returns "boolean"
typeof x                      // Returns "undefined" (if x has no value)
```

# JavaScript Data Types

The `typeof` operator can return one of two complex types:

- `function`
- `object`

The `typeof` operator returns "object" for objects, arrays, and null.

The `typeof` operator does not return "object" for functions.

```
typeof {name:'John', age:34} // Returns "object"
typeof [1,2,3,4]      // Returns "object" (not "array", see note below)
typeof null         // Returns "object"
typeof function myFunc(){} // Returns "function"
```

The `typeof` operator returns "`object`" for arrays because in JavaScript arrays are objects.

# JavaScript Functions

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

```
function myFunction(p1, p2) {
  return p1 * p2;   // The function returns the product of p1 and p2
}
```

# JavaScript Functions

A JavaScript function is defined with the `function` keyword, followed by a **name**, followed by parentheses **()**.

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
**(*parameter1, parameter2, ...*)**

The code to be executed, by the function, is placed inside curly brackets: **{}**

Function **parameters** are listed inside the parentheses () in the function definition.

Function **arguments** are the **values** received by the function when it is invoked.

Inside the function, the arguments (the parameters) behave as local variables.

A Function is much the same as a Procedure or a Subroutine, in other programming languages.

# JavaScript Functions

The code inside the function will execute when "something" **invokes** (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

You will learn a lot more about function invocation later in this session.

# JavaScript Functions

When JavaScript reaches a `return` statement, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

Functions often compute a **return value**. The return value is "returned" back to the "caller":

```
var x = myFunction(4, 3);   // Function is called, return value will end
up in x

function myFunction(a, b) {
  return a * b;               // Function returns the product of a and b
}
```

# Why Functions?

You can reuse code: Define the code once, and use it many times.

You can use the same code many times with different arguments, to produce different results.

# JavaScript Functions

Variables declared within a JavaScript function, become **LOCAL** to the function.

Local variables can only be accessed from within the function.

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

```
// code here can NOT use carName

function myFunction() {
  var carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

# JavaScript Functions Exercise

*https://www.w3schools.com/js/exercise_js.asp?filename=exercise_js_functions1*

New Horizons Computer Learning Centers

UPGRADING PEOPLE EVERY DAY

# JavaScript Objects

In real life, a car is an **object**.

A car has **properties** like weight and color, and **methods** like start and stop.

All cars have the same **properties**, but the property **values** differ from car to car (weight, color, brand).

All cars have the same **methods**, but the methods are performed **at different times**.

# JavaScript Objects

You have already learned that JavaScript variables are containers for data values.

This code assigns a **simple value** (Fiat) to a **variable** named car:

```
var car = "Fiat";
```

Objects are variables too. But objects can contain many values.

This code assigns **many values** (Fiat, 500, white) to a **variable** named car:

```
var car = {type:"Fiat", model:"500", color:"white"};
```

The values are written as **name:value** pairs (name and value separated by a colon).

JavaScript objects are containers for **named values** called properties or methods.

# JavaScript Objects

You define (and create) a JavaScript object with an object literal.

Spaces and line breaks are not important. An object definition can span multiple lines:

```
var person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

# JavaScript Objects

The **name:values** pairs in JavaScript objects are called **properties**.

You can access object properties in two ways:

```
objectName.propertyName
```

or

```
objectName["propertyName"]
```

# JavaScript Objects

Objects can also have **methods**.

Methods are **actions** that can be performed on objects.

Methods are stored in properties as **function definitions**.

A method is a function stored as a property.

```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

# JavaScript Objects

In a function definition, `this` refers to the "owner" of the function.

In the example above, `this` is the **person object** that "owns" the `fullName` function.

In other words, `this.firstName` means the `firstName` property of **this object**.

# JavaScript Objects

You access an object method with the following syntax:

*objectName.methodName()*

```
name = person.fullName();
```

If you access a method **without** the () parentheses, it will return the **function definition**:

```
name = person.fullName;
```

# JavaScript Objects

When a JavaScript variable is declared with the keyword "new", the variable is created as an object:

```
var x = new String();     // Declares x as a String object
var y = new Number();     // Declares y as a Number object
var z = new Boolean();    // Declares z as a Boolean object
```

Avoid `String`, `Number`, and `Boolean` objects. They complicate your code and slow down execution speed.

# JavaScript Objects Exercise

*https://www.w3schools.com/js/exercise_js.asp?filename=exercise_js_objects1*

# JavaScript Events

HTML events are **"things"** that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

# JavaScript Events

Often, when events happen, you may want to do something.

JavaScript lets you execute code when events are detected.

HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

With single quotes or double quotes:

```
<element event='some JavaScript'>
<element event="some JavaScript">
```

# JavaScript Events

In the following example, an `onclick` attribute (with code), is added to a `<button>` element:

```
<button onclick="document.getElementById('demo').innerHTML =
Date()">The time is?</button>
```

In the example above, the JavaScript code changes the content of the element with id="demo".

# JavaScript Events

In the next example, the code changes the content of its own element (using `this.innerHTML`):

```
<button onclick="this.innerHTML = Date()">The time
is?</button>
```

JavaScript code is often several lines long. It is more common to see event attributes calling functions:

```
<button onclick="displayDate()">The time is?</button>
```

# JavaScript Events

Here is a list of some common HTML events:

| Event | Description |
| --- | --- |
| onchange | An HTML element has been changed |
| onclick | The user clicks an HTML element |
| onmouseover | The user moves the mouse over an HTML element |
| onmouseout | The user moves the mouse away from an HTML element |
| onkeydown | The user pushes a keyboard key |
| onload | The browser has finished loading the page |

Much longer list here: https://www.w3schools.com/jsref/dom_obj_event.asp

# JavaScript Events

Event handlers can be used to handle and verify user input, user actions, and browser actions:

- Things that should be done every time a page loads
- Things that should be done when the page is closed
- Action that should be performed when a user clicks a button
- Content that should be verified when a user inputs data
- And more ...

Many different methods can be used to let JavaScript work with events:

- HTML event attributes can execute JavaScript code directly
- HTML event attributes can call JavaScript functions
- You can assign your own event handler functions to HTML elements
- You can prevent events from being sent or being handled
- And more ...

# JavaScript Events Exercise

*https://www.w3schools.com/js/exercise_js.asp?filename=exercise_js_events1*

New Horizons Computer Learning Centers

UPGRADING PEOPLE EVERY DAY

# JavaScript Strings

JavaScript strings are used for storing and manipulating text.

To find the length of a string, use the built-in `length` property:

```
var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
var sln = txt.length;
```

# JavaScript Strings

Because strings must be written within quotes, JavaScript will misunderstand this string:

```
var x = "We are the so-called "Vikings" from the north.";
```

The string will be chopped to "We are the so-called ".

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

| Code | Result | Description |
|---|---|---|
| \' | ' | Single quote |
| \" | " | Double quote |
| \\ | \ | Backslash |

# JavaScript Strings

The sequence \" inserts a double quote in a string:

```
var x = "We are the so-called \"Vikings\" from the north.";
```

The sequence \' inserts a single quote in a string:

```
var x = 'It\'s alright.';
```

# JavaScript Strings

Six other escape sequences are valid in JavaScript:

| Code | Result |
|---|---|
| \b | Backspace |
| \f | Form Feed |
| \n | New Line |
| \r | Carriage Return |
| \t | Horizontal Tabulator |
| \v | Vertical Tabulator |

# JavaScript Strings

For best readability, programmers often like to avoid code lines longer than 80 characters.

If a JavaScript statement does not fit on one line, the best place to break it is after an operator:

```
document.getElementById("demo").innerHTML =
"Hello Dolly!";
```

You can also break up a code line **within a text string**:

```
document.getElementById("demo").innerHTML = "Hello " +
"Dolly!";
```

# JavaScript Strings Exercise

*https://www.w3schools.com/js/exercise_js.asp?filename=exercise_js_strings1*

New Horizons Computer Learning Centers

UPGRADING PEOPLE EVERY DAY

# JavaScript Strings

The `indexOf()` method returns the index of (the position of) the `first` occurrence of a specified text in a string:

```
var str = "Please locate where 'locate' occurs!";
var pos = str.indexOf("locate");
```

JavaScript counts positions from zero.

0 is the first position in a string, 1 is the second, 2 is the third ...

# JavaScript Strings

The `lastIndexOf()` method returns the index of the **last** occurrence of a specified text in a string:

```
var str = "Please locate where 'locate' occurs!";
var pos = str.lastIndexOf("locate");
```

Both `indexOf()`, and `lastIndexOf()` return -1 if the text is not found.

# JavaScript Strings

Both methods accept a second parameter as the starting position for the search:

```
var str = "Please locate where 'locate' occurs!";
var pos = str.indexOf("locate", 15);
```

The `lastIndexOf()` methods searches backwards (from the end to the beginning), meaning: if the second parameter is `15`, the search starts at position 15, and searches to the beginning of the string.

# JavaScript Strings

The `search()` method searches a string for a specified value and returns the position of the match:

```
var str = "Please locate where 'locate' occurs!";
var pos = str.search("locate");
```

# JavaScript Strings

The two methods, `indexOf()` and `search()`, are **equal?**

They accept the same arguments (parameters), and return the same value?

The two methods are **NOT** equal. These are the differences:

- The `search()` method cannot take a second start position argument.
- The `indexOf()` method cannot take powerful search values (regular expressions).

# JavaScript Strings

There are 3 methods for extracting a part of a string:

- `slice(start, end)`
- `substring(start, end)`
- `substr(start, length)`

# JavaScript Strings

`slice()` extracts a part of a string and returns the extracted part in a new string.

The method takes 2 parameters: the start position, and the end position (end not included).

This example slices out a portion of a string from position 7 to position 12 (13-1):

```
var str = "Apple, Banana, Kiwi";
var res = str.slice(7, 13); //RESULT: Banana
```

# JavaScript Strings

If a parameter is negative, the position is counted from the end of the string.

This example slices out a portion of a string from position -12 to position -6:

```
var str = "Apple, Banana, Kiwi";
var res = str.slice(-12, -6); //RESULT: Banana
```

you omit the second parameter, the method will slice out the rest of the string:

```
var res = str.slice(7); //RESULT: Banana, Kiwi
```

or, counting from the end:

```
var res = str.slice(-12); //RESULT: Banana, Kiwi
```

# JavaScript Strings

`substring()` is similar to `slice()`.

The difference is that `substring()` cannot accept negative indexes.

```
var str = "Apple, Banana, Kiwi";
var res = str.substring(7, 13); //RESULT: Banana
```

# JavaScript Strings

`substr()` is similar to `slice()`.

The difference is that the second parameter specifies the **length** of the extracted part.

```
var str = "Apple, Banana, Kiwi";
var res = str.substr(7, 6); //RESULT: Banana
```

# JavaScript Strings

The `replace()` method replaces a specified value with another value in a string:

```
str = "Please visit Micro
soft!";
var n = str.replace("Microsoft", "W3Schools");
```

The `replace()` method does not change the string it is called on. It returns a new string.

# JavaScript Strings

By default, the `replace()` method replaces **only the first** match:

```
str = "Please visit Microsoft and Microsoft!";
var n = str.replace("Microsoft", "W3Schools");
```

By default, the `replace()` method is case sensitive. Writing MICROSOFT (with upper-case) will not work:

```
str = "Please visit Microsoft!";
var n = str.replace("MICROSOFT", "W3Schools");
```

# JavaScript Strings

To replace case insensitive, use a **regular expression** with an /i flag (insensitive):

```
str = "Please visit Microsoft!";
var n = str.replace(/MICROSOFT/i, "W3Schools");
```

Note that regular expressions are written without quotes.
To replace all matches, use a **regular expression** with a /g flag (global match):

```
str = "Please visit Microsoft and Microsoft!";
var n = str.replace(/Microsoft/g, "W3Schools");
```

# JavaScript Strings

A string is converted to upper case with `toUpperCase()`:

```
var text1 = "Hello World!";   // String
var text2 = text1.toUpperCase(); // text2 is text1 converted to upper
```

A string is converted to lower case with `toLowerCase()`:

```
var text1 = "Hello World!";   // String
var text2 = text1.toLowerCase(); // text2 is text1 converted to lower
```

# JavaScript Strings

`concat()` joins two or more strings:

```
var text1 = "Hello";
var text2 = "World";
var text3 = text1.concat(" ", text2);
```

The `concat()` method can be used instead of the plus operator. These two lines do the same:

```
var text = "Hello" + " " + "World!";
var text = "Hello".concat(" ", "World!");
```

All string methods return a new string. They don't modify the original string. Formally said: Strings are immutable: Strings cannot be changed, only replaced.

# JavaScript Strings

The `trim()` method removes whitespace from both sides of a string:

```
var str = "    Hello World!    ";
alert(str.trim());
```

The `trim()` method is not supported in Internet Explorer 8 or lower.

# JavaScript Strings

ECMAScript 2017 added two String methods: `padStart` and `padEnd` to support padding at the beginning and at the end of a string.

```
let str = "5";
str = str.padStart(4,0);
// result is 0005
```

```
let str = "5";
str = str.padEnd(4,0);
// result is 5000
```

String Padding is not supported in Internet Explorer.

# JavaScript Strings

The `charAt()` method returns the character at a specified index (position) in a string:

```
var str = "HELLO WORLD";
str.charAt(0);      // returns H
```

# JavaScript Strings

The `charCodeAt()` method returns the unicode of the character at a specified index in a string:

The method returns a UTF-16 code (an integer between 0 and 65535).

```
var str = "HELLO WORLD";
str.charCodeAt(0);          // returns 72
```

# JavaScript Strings

ECMAScript 5 (2009) allows property access [ ] on strings:

```
var str = "HELLO WORLD";
str[0];                    // returns H
```

Property access might be a little **unpredictable:**

- It does not work in Internet Explorer 7 or earlier
- It makes strings look like arrays (but they are not)
- If no character is found, [ ] returns undefined, while charAt() returns an empty string.
- It is read only. str[0] = "A" gives no error (but does not work!)

# JavaScript Strings

A string can be converted to an array with the `split()` method:

```
var txt = "a,b,c,d,e";  // String
txt.split(",");     // Split on commas
txt.split(" ");     // Split on spaces
txt.split("|");     // Split on pipe
```

If the separator is omitted, the returned array will contain the whole string in index [0].

If the separator is "", the returned array will be an array of single characters:

```
var txt = "Hello";   // String
txt.split("");      // Split in characters
```

# JavaScript String Methods Exercise

*https://www.w3schools.com/js/exercise_js.asp?filename=exercise_js_string_methods1*

# JavaScript Numbers

Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc.

JavaScript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard.

This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63.

# JavaScript Numbers

Integers (numbers without a period or exponent notation) are accurate up to 15 digits.

```
var x = 999999999999999;   // x will be 999999999999999
var y = 9999999999999999;  // y will be 10000000000000000
```

The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate:

```
var x = 0.2 + 0.1;         // x will be 0.30000000000000004
```

To solve the problem above, it helps to multiply and divide:

```
var x = (0.2 * 10 + 0.1 * 10) / 10;       // x will be 0.3
```

# JavaScript Numbers

`NaN` is a JavaScript reserved word indicating that a number is not a legal number. Trying to do arithmetic with a non-numeric string will result in `NaN` (Not a Number):

```
var x = 100 / "Apple"; // x will be NaN (Not a Number)
```

However, if the string contains a numeric value , the result will be a number:

```
var x = 100 / "10";  // x will be 10
```

Watch out for `NaN`. If you use `NaN` in a mathematical operation, the result will also be `NaN`:

```
var x = NaN;
var y = 5;
var z = x + y;    // z will be NaN
```

# JavaScript Numbers

`Infinity` (or `-Infinity`) is the value JavaScript will return if you calculate a number outside the largest possible number.

```
var myNumber = 2;
while (myNumber != Infinity) { // Execute until Infinity
 myNumber = myNumber * myNumber;
}
```

Division by 0 (zero) also generates `Infinity`:

```
var x = 2 / 0;   // x will be Infinity
var y = -2 / 0;   // y will be –Infinity
```

`Infinity` is a number: `typeof Infinity` returns `number`.

```
typeof Infinity;  // returns "number"
```

# JavaScript Numbers

JavaScript interprets numeric constants as hexadecimal if they are preceded by 0x.

```
var x = 0xFF;    // x will be 255
```

Never write a number with a leading zero (like 07).
Some JavaScript versions interpret numbers as octal if they are written with a leading zero.

By default, JavaScript displays numbers as **base 10** decimals.

But you can use the `toString()` method to output numbers from **base 2** to **base 36**.
Hexadecimal is **base 16**. Decimal is **base 10**. Octal is **base 8**. Binary is **base 2**.

```
var myNumber = 32;
myNumber.toString(10); // returns 32
myNumber.toString(32); // returns 10
myNumber.toString(16); // returns 20
myNumber.toString(8);  // returns 40
myNumber.toString(2);  // returns 100000
```

# JavaScript Numbers

Normally JavaScript numbers are primitive values created from literals:

```
var x = 123;
```

But numbers can also be defined as objects with the keyword `new`:

```
var y = new Number(123);
```

# JavaScript Numbers

Do not create Number objects. It slows down execution speed.
The `new` keyword complicates the code. This can produce some unexpected results:

When using the `===` operator, equal numbers are not equal, because the `===` operator expects equality in both type and value.

```
var x = 500;
var y = new Number(500);

// (x === y) is false because x and y have different types
```

Or even worse. Objects cannot be compared:

```
var x = new Number(500);
var y = new Number(500);

// (x == y) is false because objects cannot be compared
```

# JavaScript Numbers

Primitive values (like 3.14 or 2014), cannot have properties and methods (because they are not objects).

But with JavaScript, methods and properties are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties.

# JavaScript Numbers

`toExponential()` returns a string, with a number rounded and written using exponential notation.

A parameter defines the number of characters behind the decimal point:

```
var x = 9.656;
x.toExponential(2);  // returns 9.66e+0
x.toExponential(4);  // returns 9.6560e+0
x.toExponential(6);  // returns 9.656000e+0
```

The parameter is optional. If you don't specify it, JavaScript will not round the number.

# JavaScript Numbers

`toFixed()` returns a string, with the number written with a specified number of decimals:

```
var x = 9.656;
x.toFixed(0);      // returns 10
x.toFixed(2);      // returns 9.66
x.toFixed(4);      // returns 9.6560
x.toFixed(6);      // returns 9.656000
```

`toFixed(2)` is perfect for working with money.

# JavaScript Numbers

`toPrecision()` returns a string, with a number written with a specified length:

```
var x = 9.656;
x.toPrecision();    // returns 9.656
x.toPrecision(2);   // returns 9.7
x.toPrecision(4);   // returns 9.656
x.toPrecision(6);   // returns 9.65600
```

# JavaScript Numbers

`valueOf()` returns a number as a number.

```
var x = 123;
x.valueOf();      // returns 123 from variable x
(123).valueOf();    // returns 123 from literal 123
(100 + 23).valueOf(); // returns 123 from expression 100 + 23
```

In JavaScript, a number can be a primitive value (typeof = number) or an object (typeof = object).

The `valueOf()` method is used internally in JavaScript to convert Number objects to primitive values.

There is no reason to use it in your code.

All JavaScript data types have a `valueOf()` and a `toString()` method.

# JavaScript Numbers

There are 3 JavaScript methods that can be used to convert variables to numbers:

`Number()` can be used to convert JavaScript variables to numbers:

```
Number(true);      // returns 1
Number(false);     // returns 0
Number("10");      // returns 10
Number("  10");    // returns 10
Number("10  ");    // returns 10
Number(" 10  ");   // returns 10
Number("10.33");    // returns 10.33
Number("10,33");    // returns NaN
Number("10 33");   // returns NaN
Number("John");    // returns NaN
```

If the number cannot be converted, `NaN` (Not a Number) is returned.

# JavaScript Numbers

`Number()` can also convert a date to a number:

```
Number(new Date("2017-09-30"));   // returns 1506729600000
```

The `Number()` method above returns the number of milliseconds since 1.1.1970.

# JavaScript Numbers

`parseInt()` parses a string and returns a whole number. Spaces are allowed. Only the first number is returned:

```
parseInt("10");    // returns 10
parseInt("10.33");   // returns 10
parseInt("10 20 30"); // returns 10
parseInt("10 years"); // returns 10
parseInt("years 10"); // returns NaN
```

# JavaScript Numbers

`parseFloat()` parses a string and returns a number. Spaces are allowed. Only the first number is returned:

```
parseFloat("10");    // returns 10
parseFloat("10.33");  // returns 10.33
parseFloat("10 20 30"); // returns 10
parseFloat("10 years"); // returns 10
parseFloat("years 10"); // returns NaN
```

New Horizons®
Computer Learning Centers

UPGRADING
PEOPLE
EVERY DAY

# *Thank You For Your Time*