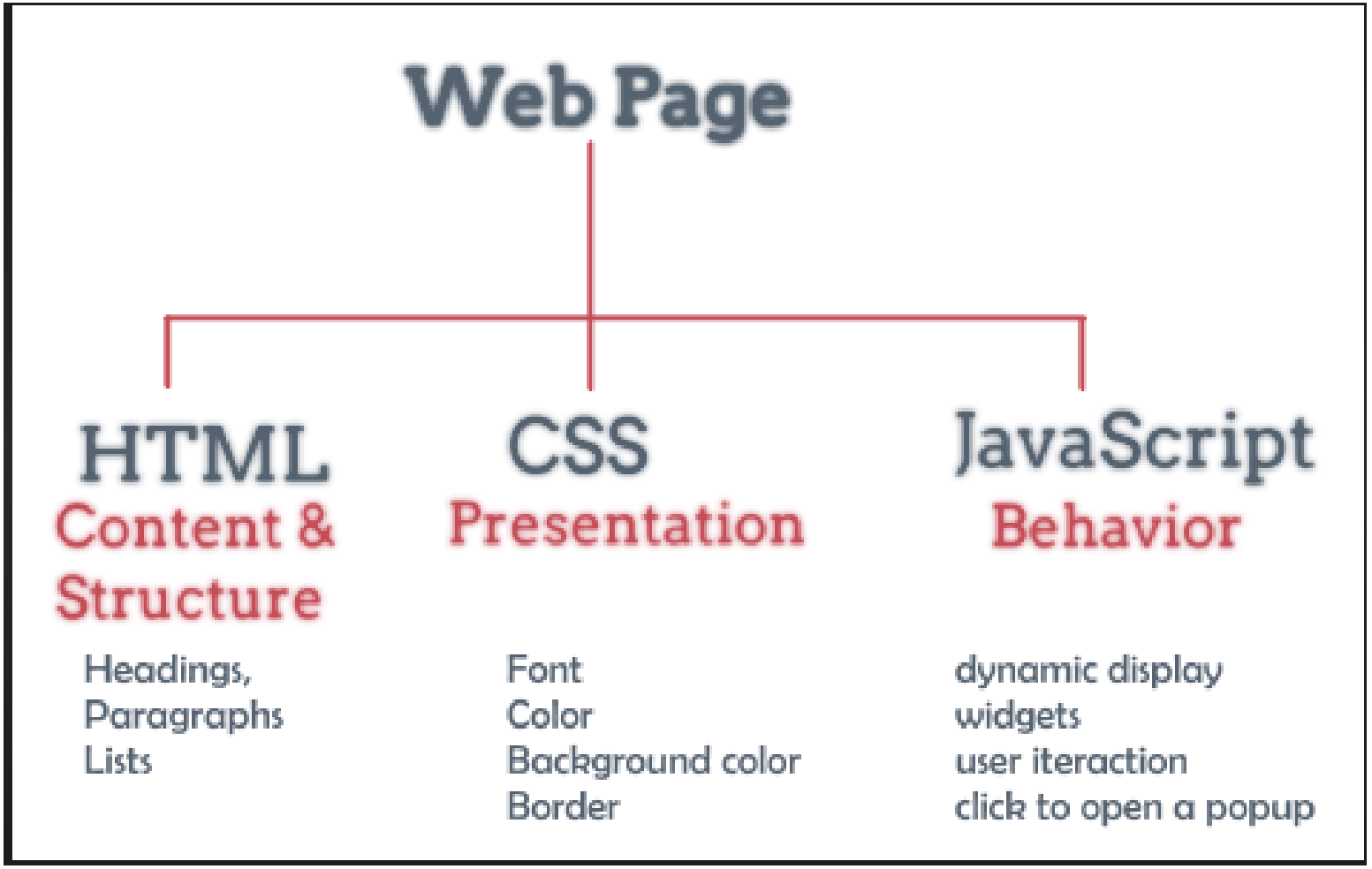
INTRODUCTION TO JAVASCRIPT

Presented by
Eng./Abanoub Nabil
Teaching assistant at ITI

JavaScript …


Web Page
HTML
Content & Structure
Headings,
Paragraphs
Lists
CSS
Presentation
Font
Color
Background color
Border
JavaScript
Behavior
dynamic display
widgets
user iteraction
click to open a popup

JavaScript Where To?

- We can Write JavaScript:

  1. Anywhere in the html file between script tags.

```
<html>
   <head>
      <title>A Simple Document</title>
      <script >
         document.write ("Hello world");
      </ script >
   </head>
   <body>
      <p>Page content</p>
      < script >
         document.write (" welcome to JavaScript world");
      </ script >
   </body>
</html>
```

JavaScript Where To?

- We can Write JavaScript:

  2. In an external file and refer to it using the src attribute.

myWebPage.html

```
<HEAD>
  <TITLE>A Simple Document</TITLE>
 <script src= "myJSFile.js" >  </script>
</HEAD>

<BODY>
  We can refer to JavaScript in another file.
  < script >
          dosomething();
  </ script >

</BODY>
```

myJSFile.js

```
function dosomething()
{
    alert ("Hello ");
}
.
.
.
.
```

External JavaScript Advantages

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.

- **To add several script files to one page - use several script tags:**

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

Meet the Console Tab of Web Development Tools

- Web development tools allow you to **test and debug** the JavaScript code.
- Web development tools are often called **devtools**.
- Modern web browsers such as Google Chrome, Firefox, Edge, Safari, and Opera **provide the devtools as built-in features**.
- Generally, **devtools allow** you to work with a variety of web technologies such as HTML, CSS, DOM, and JavaScript.

Meet the Console Tab of Web Development Tools

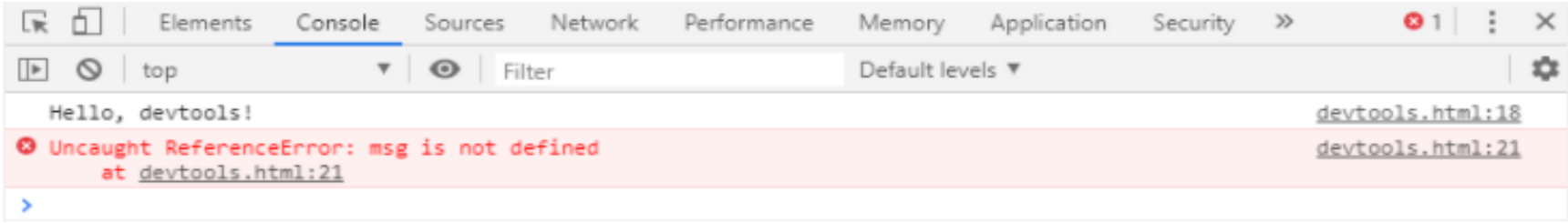
- **Google Chrome**
- Suppose that you have the following piece of code

```
<script>
    console.log('Hello, devtools!');

    // the following code causes an error
    let greeting = msg;
</script>
```

- Run it and open console.

Meet the Console Tab of Web Development Tools



- The first message is **'Hello, DevTools!'** which is the output of the following command:

```
console.log('Hello, DevTools!');
```

- The second message that appeared on the Console tab is an error.
- This is because the variable msg has not been defined in the code but was referenced in the assignment.
- For more details about other browsers:-
- https://www.javascripttutorial.net/web-development-tools/

JavaScript Hello World Example

JavaScript can "display" data in different ways:

1- Writing into an HTML element, using **innerHTML**.

2- Writing into the HTML output using **document.write().**

3- Writing into an alert box, using **window.alert().**

4- Writing into the browser console, using **console.log().**

DEMO !!!!

JavaScript Comments.

- JavaScript supports **both** single-line and block comments.
- A single-line comment starts with two forward-slash characters (//), for example:

```
// this is a single-line comment
```

- A block comment starts with a forward slash and asterisk (/*) and ends with the opposite (*/) as follows

```
/*
* This is a block comment that can
* span multiple lines
*/
```

JavaScript is case-sensitive

- Everything in JavaScript
  including **variables**, **function** names, **class** names, and operators are **case-sensitive.**

- It means that **counter** and **Counter** variables are different.

- Likewise, you cannot use **instanceof** as the name of a function because it is a keyword. However, **instanceOf** is a valid function name.

Identifiers

- An **identifier** is the name of a variable, function, parameter, or class. An identifier consists of one or more characters in the following format:

  1- **The first character** must be a letter (a-z, or A-Z), an underscore(_), or a dollar sign ($).

  2-**The other characters** can be letters (a-z, A-Z), numbers (0-9), underscores (_), and dollar signs ($).

- **It is a good practice** to use **camel case** for the identifiers, meaning that the first letter is lowercase, and each additional word starts with a capital letter **as the following examples**

-name -firstName -studentAge -calculateSum -counter

Statements (Semicolon)

- Although JavaScript **does not require** to end a statement with a semicolon (;), **it is recommended** to always use the semicolon to end a statement.

- The reason is that the **semicolon will make your code more readable** and helps you avoid many issues that you may encounter.

- **In addition**, you may need to ***combine and compress*** the JavaScript code before deploying it to the production environment to remove extra white space to save the bandwidth; without the semicolons, you will have the syntax errors.

```
var a = 10;
var b = 20;
```

Keywords & Reserved words

- JavaScript defines a list of keywords and reserved words that have special uses.
- You cannot use the keywords and reserved words as the identifiers.
- The list of JavaScript keywords and reserved words is as follows:

abstract	arguments	await	boolean
break	byte	case	catch
char	class	const	continue
debugger	default	delete	do
double	else	enum	eval

Keywords & Reserved words (cont.)

export	extends	false	final
finally	float	for	function
goto	if	implements	import
in	instanceof	int	interface
let	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

JavaScript variables

- A **JavaScript variable** is simply a name of storage location.
- There are two types of variables in JavaScript : **local variable and global variable.**
- JavaScript variables are **loosely typed**, that is to say, variables can hold values with any type of data. Variables are just named placeholders for values.
- **Declare JavaScript variables using var keyword**
- To declare a variable, you use the **var** keyword followed by the **variable name** as follows:
-

```
var message;
```

- A variable name can be any valid **identifier**. The message variable is declared and hold a special value **undefined**.

JavaScript variables (cont.)

- After declaring a variable, you can give a value to it as the following.

```
message = "Hello";
```

- You can do the previous two steps in one step as the following:

```
var variableName = value;
```

- Example

```
var message = "Hello";
```

JavaScript variables (cont.)

You can declare two or more variables using one statement, each variable declaration is separated by a comma (,) as follows:

```
var message = "Hello",
    counter = 100;
```

As mentioned earlier, you can store a number in the `message` variable as the following example though it is not recommended.

```
message = 100;
```

JavaScript variables (cont.)

- **Undefined vs. undeclared variables**
- It's important to distinguish between **undefined** and **undeclared** variables.

- An **undefined variable** is a variable that has been declared.
- Because we have not assigned it a value, the variable used the **undefined as its initial value.**

- In contrast, an undeclared variable is the variable that has not been declared.

- See the following example:

JavaScript variables (cont.)

```
var message;


console.log(message); // undefined
console.log(counter); // ReferenceError: counter is not defined
```

In this example, the message variable is declared but not initialized therefore its value is **undefined** whereas the counter variable has not been declared hence accessing it causes a **ReferenceError**.

JavaScript data types

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

- JavaScript provides different **data types** to hold different types of values. There are two types of data types in JavaScript.

  1-Primitive data type
  2-Non-primitive (reference) data type

-

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

JavaScript data types

JavaScript primitive data types

There are five types of primitive data types in JavaScript. They are as follows:

Data Type	Description
String	represents sequence of characters e.g. "hello"
Number	represents numeric values e.g. 100
Boolean	represents boolean value either false or true
Undefined	represents undefined value
Null	represents null i.e. no value at all

JavaScript data types

JavaScript non-primitive data types

The non-primitive data types are as follows:

Data Type	Description
Object	represents instance through which we can access members
Array	represents group of similar values
RegExp	represents regular expression

We will have great discussion on each data type later.

JavaScript data types (cont.)

JavaScript

Creating a variable in JavaScript is called "declaring" a variable.

You declare a JavaScript variable with the `var` keyword:

```
var carName;
```

After the declaration, the variable has no value (technically it has the value of `undefined`).

To **assign** a value to the variable, use the equal sign:

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

You can also assign a value to the variable when you declare it:

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

JavaScript data types (cont.)

- 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 data types (cont.)

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

Example

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

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

Example

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

JavaScript data types (cont.)

Also try this:

Example

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

Now try this:

Example

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

JavaScript data types (cont.)

- To get the current type of the value of a variable, you use the **typeof** operator:

```
typeof ""              // Returns "string"
typeof "John"          // Returns "string"
typeof "John Doe"      // Returns "string"
```

JavaScript data types (cont.)

- 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 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
```

Difference Between Undefined and Null

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

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

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

JavaScript arithmetic operations

Arithmetic operators perform arithmetic on numbers (literals or variables).

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

JavaScript 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 Logical Operators

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

JavaScript Assignment Operators

Operator	Meaning	Description
a = b	a = b	Assigns the value of b to a.
a += b	a = a + b	Assigns the result of a plus b to a.
a -= b	a = a - b	Assigns the result of a minus b to a.
a *= b	a = a * b	Assigns the result of a times b to a.
a /= b	a = a / b	Assigns the result of a divided by b to a.
a %= b	a = a % b	Assigns the result of a modulo b to a.

Program flow (if –else)

- **Conditional statements** are used to perform different actions based on different conditions.

Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

Program flow (if –else)

The `if` statement is probably one of the most frequently used statements in JavaScript. The `if` statement executes a statement or block of code if a condition is satisfied. The following is the simple form of the `if` statement:

```
if( condition )
   statement;
```

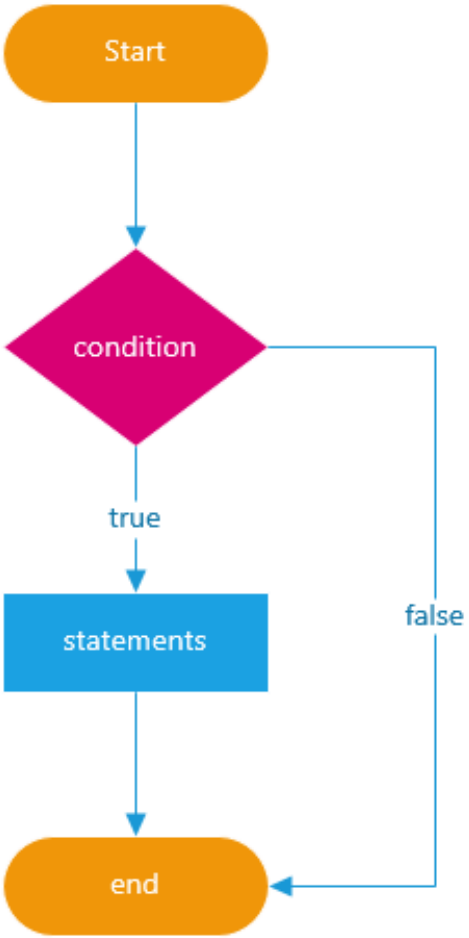
The `condition` can be any valid expression. In general, the condition evaluates to a Boolean value, either `true` or `false`.

In case the `condition` evaluates to a non-Boolean value, JavaScript implicitly converts its result into a Boolean value by calling the `Boolean()` function.

If the `condition` evaluates to `true`, the `statement` is executed. Otherwise, the control is passed to the next statement that follows the `if` statement.

Program flow (if –else)

The following flowchart illustrates the `if` statement.



Program flow (if –else)

```
var x = 25;
if( x > 10 )
    console.log('x is greater than 10');
```

- This example first initializes the variable x to 25. The x > 10 expression evaluates to true, therefore the script shows a message in the console window.

- In case you have more than one statement to execute, you need to use curly braces as follows:

Program flow (if –else)

JavaScript

In case you have more than one statement to execute, you need to use curly braces as follows:

```
if( condition ) {
    // statements
}
```

It is a good programming practice to always use the curly braces even if there is only one statement to be executed. This helps the code easier to read and avoid many confusions.

See the following example:

```
if( condition )
    statement_1
    statement_2;
```

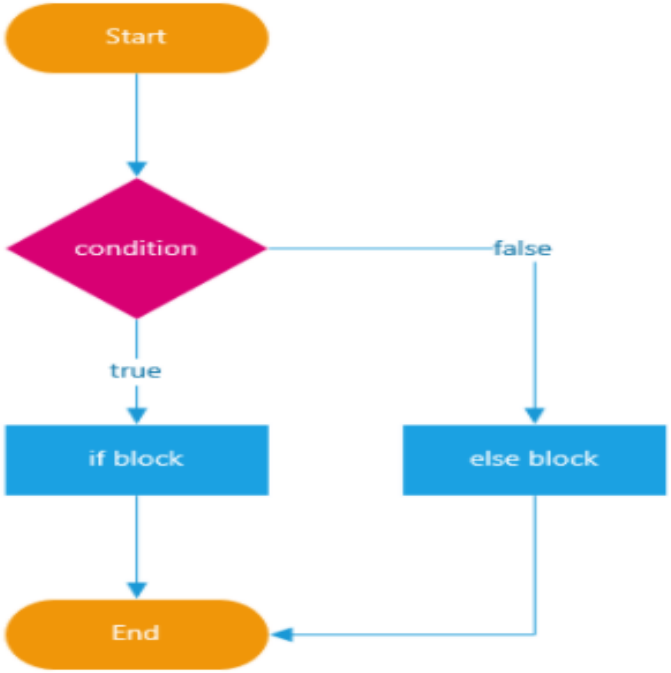
If you don't use the curly braces, it will be difficult to see that the `statement_2` does not belong to the `if` block.

Program flow (if –else)

In case you want to execute another statement when the `condition` evaluates to `false`, you use the `else` as follows:

```
if( condition ) {
  // statement
} else {
  // statement (when condition evaluates to false)
}
```

The following flowchart illustrates the `if else` statement.



Program flow (if –else)

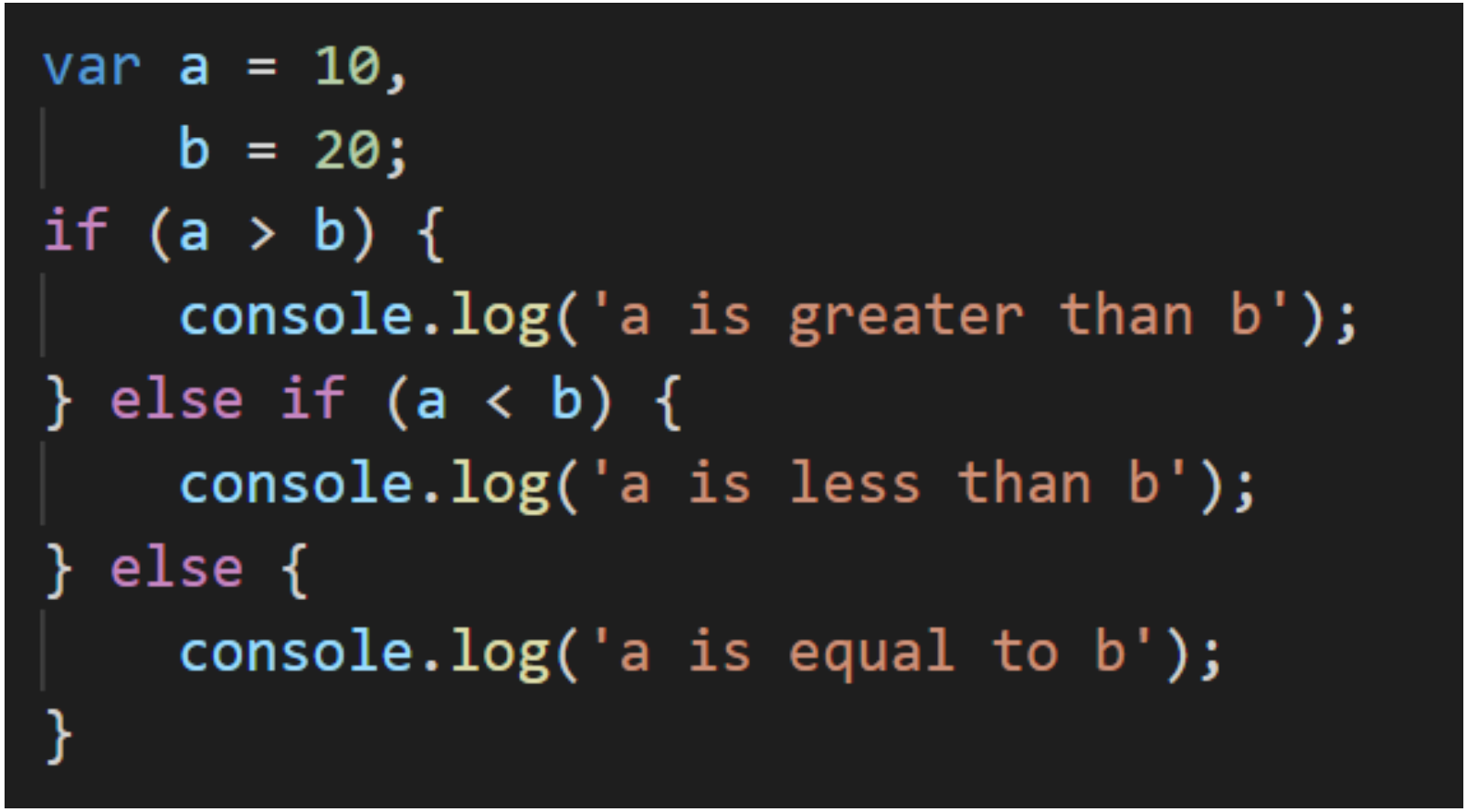
JavaScript

You can chain the `if else` statements:

```
if (condition_1) {
   // statments
} else if (condition_2) {
  // statments
} else {
   // statments
}
```

For example, the following script compares two numbers `a` and `b`, shows the corresponding message if `a` is greater than, less than, or equal to `b`.

Program flow (if –else)



Program flow (ternary operator)

JavaScript provides a conditional operator or ternary operator that can be used as a shorthand of the `if else` statement.

The following illustrates the syntax of the conditional operator.

```
condition ? expression_1 : expression_2
```

Like the `if` statement, the `condition` is an expression that evaluates to `true` or `false`.

If the condition evaluates to `true`, the operator returns the value of the `expression_1`; otherwise, it returns the value of the `expression_2`.

The following express uses the conditinal operator to return different labels for the login button based on the value of the `isLoggedIn` variable:

```
isLoggedIn ? "Logout" : "Login";
```

Typically, you assign a variable the result of the ternary operator, like this:

Program flow (ternary operator)

JavaScript

The following express uses the conditinal operator to return different labels for the login button based on the value of the `isLoggedIn` variable:

```
isLoggedIn ? "Logout" : "Login";
```

Typically, you assign a variable the result of the ternary operator, like this:

```
// only register if the age is greater than 18
var allowRegister = age > 18 ? true : false;
```

If you want to do more than a single operation per case, you need to separate operation using a comma (,) as the following example:

```
age > 18 ? (
    alert("OK, you can register."),
    redirectTo("register.html");
) : (
    stop = true,
    alert("Sorry, you are too young!")
);
```

Program flow (switch case statement)

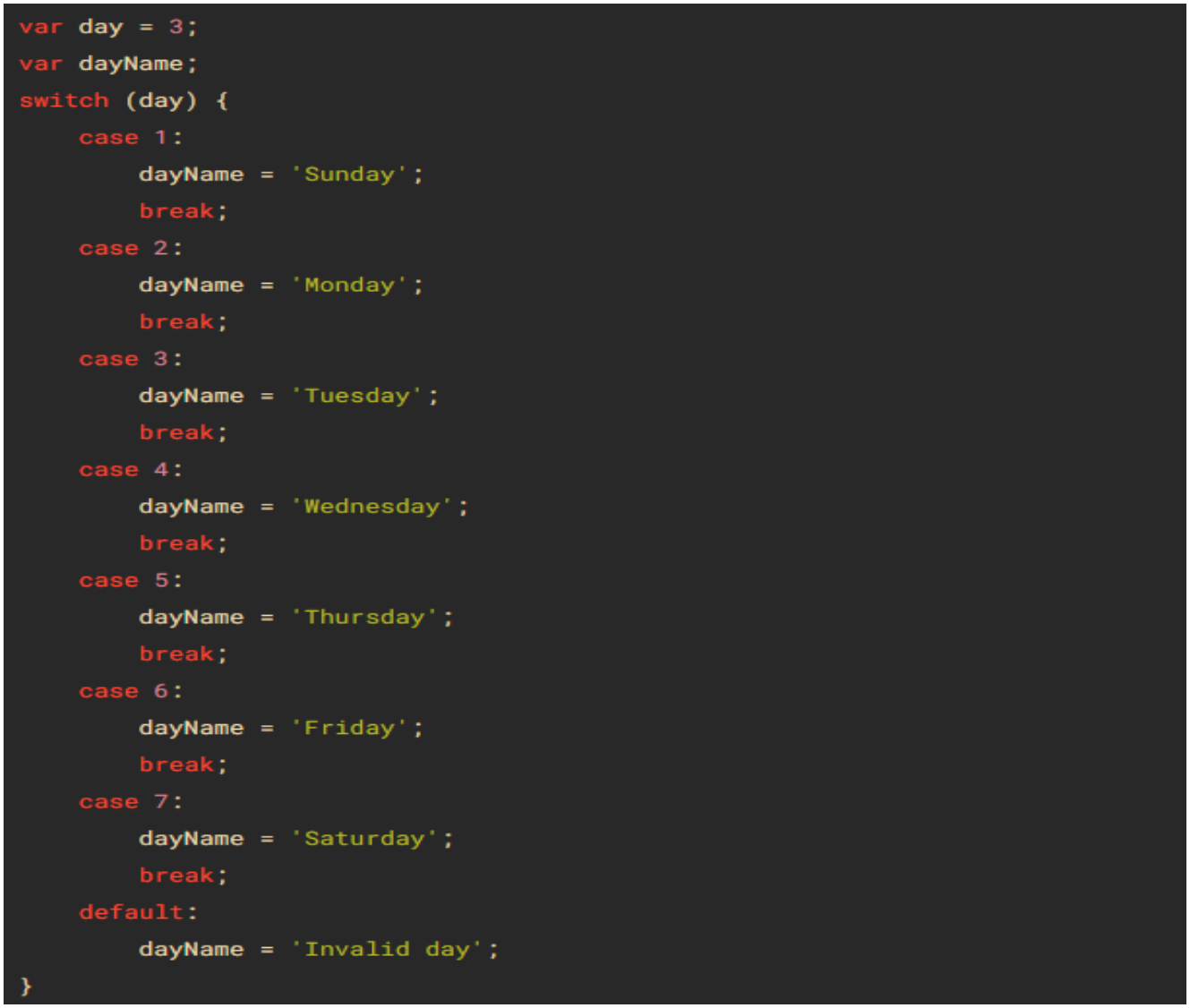
The `switch` statement is a flow-control statement that is similar to the `if else` statement. You use the `switch` statement to control the complex conditional operations.

The following illustrates the syntax of the `switch` statement:

```
switch (expression) {
    case value_1:
        statement_1;
        break;
    case value_2:
        statement_2;
        break;
    case value_3:
        statement_3;
        break;
    default:
        default_statement;
}
```

Each case in the `switch` statement executes the corresponding statement (`statement_1` , `statement_2` ,...) if the `expression` equals the value (`value_1` , `value_2` , ...).

Program flow (switch case statement)



```
var day = 3;
var dayName;
switch (day) {
    case 1:
        dayName = 'Sunday';
        break;
    case 2:
        dayName = 'Monday';
        break;
    case 3:
        dayName = 'Tuesday';
        break;
    case 4:
        dayName = 'Wednesday';
        break;
    case 5:
        dayName = 'Thursday';
        break;
    case 6:
        dayName = 'Friday';
        break;
    case 7:
        dayName = 'Saturday';
        break;
    default:
        dayName = 'Invalid day';
}
```

Program flow (for loop)

The JavaScript for loop statement allows you to create a loop with three optional expressions. The following illustrates the syntax of the for loop statement:

```
for (initialization; condition; post-expression) {
    // statements
}
```

Program flow (for loop)

JavaScript

1) initialization

The `initialization` expression initializes the loop. The initialization expression is executed only once when the loop starts. You typically use the `initialization` is to initialize a counter variable. If you use the var keyword to declare the counter variable, the variable will have either function or global scope. In other words, you can reference the counter variable after the loop ends. However, if you use the `let` keyword to declare the counter variable, the variable will have a blocked scope, which is only accessible inside the loop.

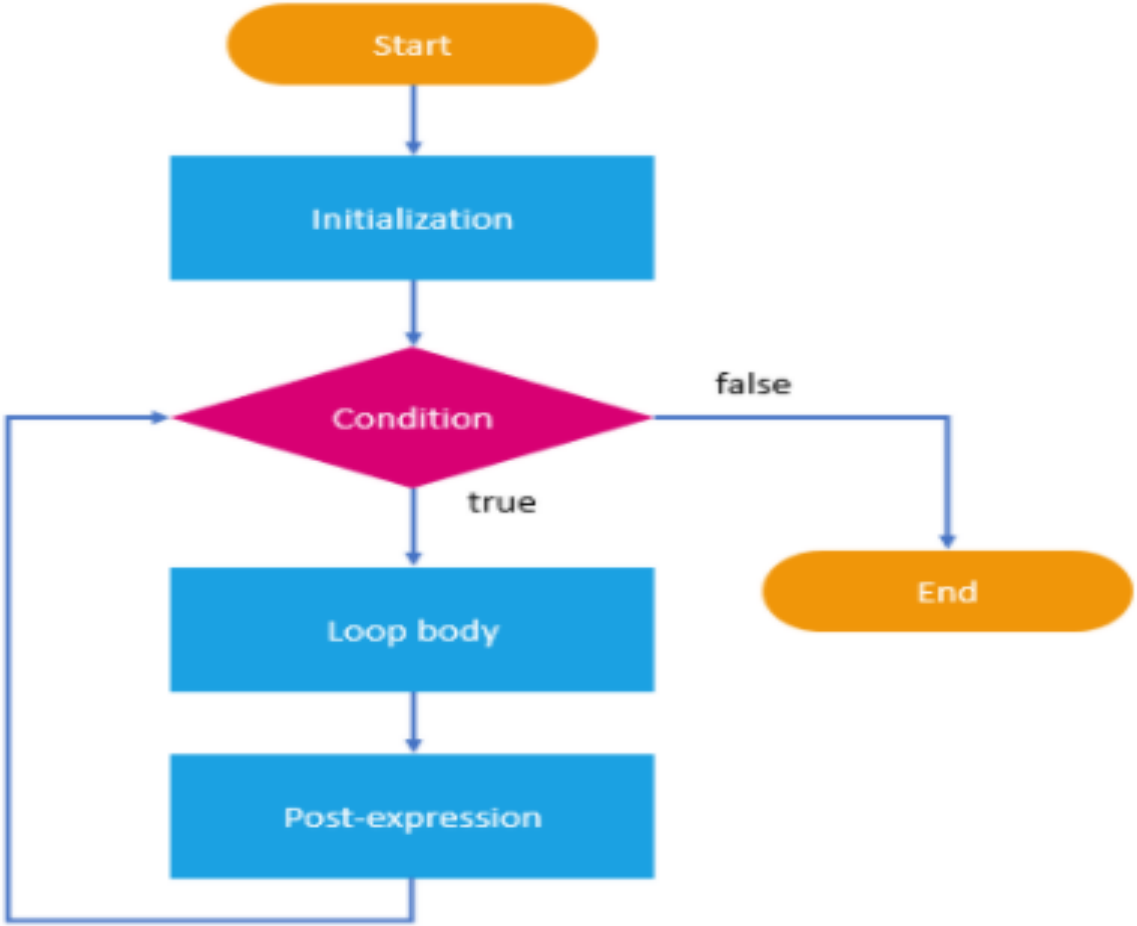
2) condition

The `condition` is an expression that is evaluated once before every iteration. The `statement` inside the loop is executed only when the `condition` evaluates to `true`. The loop is terminated if the `condition` evaluates to `false`. Note that the `condition` is optional. If you omit it, the `for` loop statement considers it as `true`.

3) post-expression

The `for` loop statement also evaluates the `post-expression` after each loop iteration. Generally, you use the `post-expression` to update the counter variable. The following flowchart illustrates the `for` loop:

Program flow (for loop)



Program flow (for loop)

JavaScript

1) Simple `for` loop examples

The following example uses the `for` loop statement that shows the numbers from 1 to 5 in the Console window.

```
for (var counter = 1; counter < 5; counter++) {
    console.log('Inside the loop:' + counter);
}
console.log('Outside the loop:' + counter);
```

Output:

```
Inside the loop:1
Inside the loop:2
Inside the loop:3
Inside the loop:4
Outside the loop:5
```

Program flow (for loop)

In the for loop, the three expressions are optional. The following shows how to use the for loop without any expressions:

```
for ( ; ; ) {
    // statements
}
```

For more details

https://www.javascripttutorial.net/javascript-for-loop/

Program flow (while Loop)

JavaScript

The JavaScript while statement creates a loop that executes a block of code **as long as the test condition evaluates to true.**

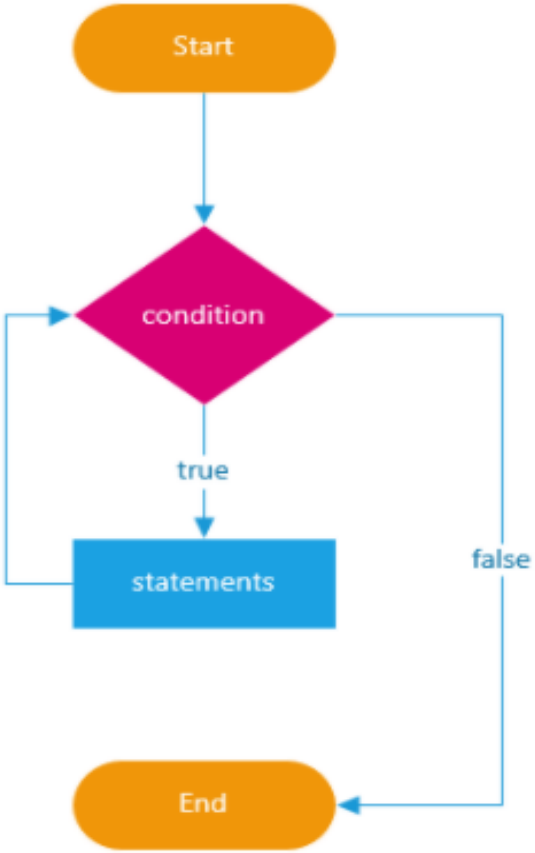
The following illustrates the syntax of the `while` statement.

```
while (expression) {
    // statement
}
```

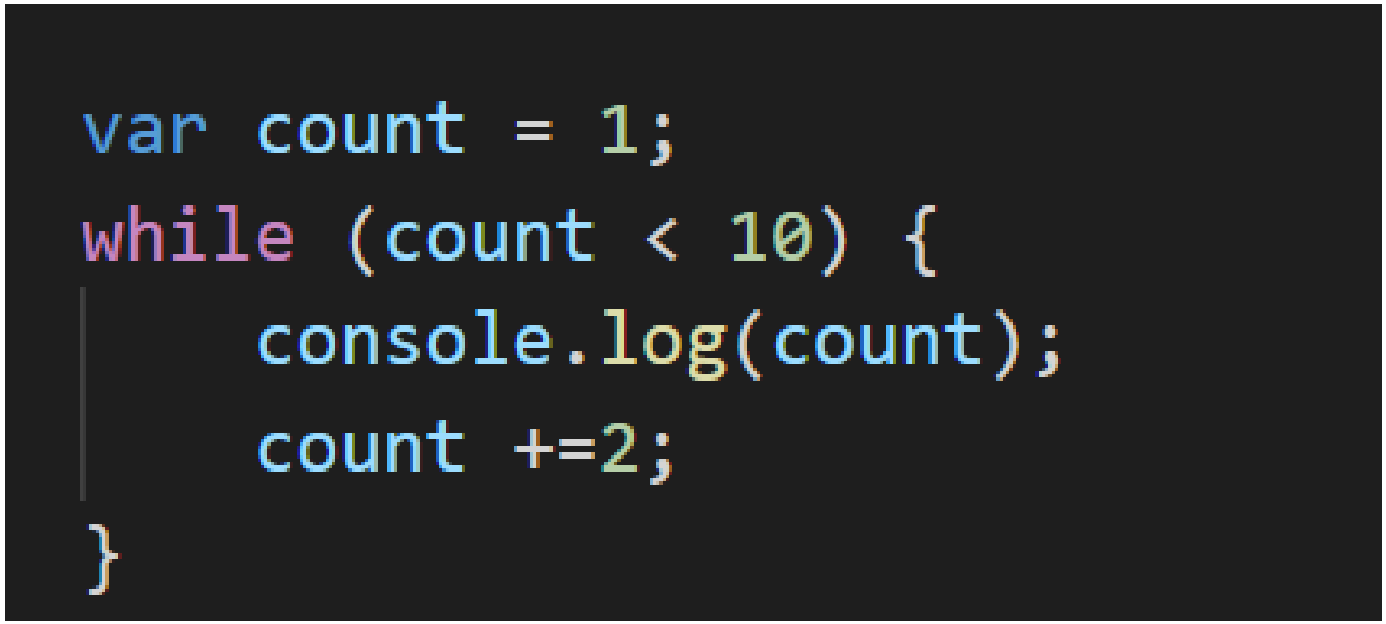
The `while` statement evaluates the `expression` before each iteration of the loop.

Program flow (while Loop)

The following flowchart illustrates the `while` loop statement:



Program flow (while Loop)



Program flow (while Loop)

JavaScript

How the script works

- First, outside of the loop, the `count` variable is set to 1.
- Second, before the first iteration begins, the `while` statement checks if `count` is less than `10` and execute the statements inside the loop body.
- Third, in each iteration, the loop increments `count` by `2` and after `5` iterations, the condition `count < 10` is no longer `true`, so the loop terminates.

The output of the script in the console window is as follows:

```
1
3
5
7
9
```

Program flow (do-while Loop)

JavaScript

The **do-while** loop statement creates a loop that executes a block of code until a test condition evaluates to false.

The following statement illustrates the syntax of the `do-while` loop:

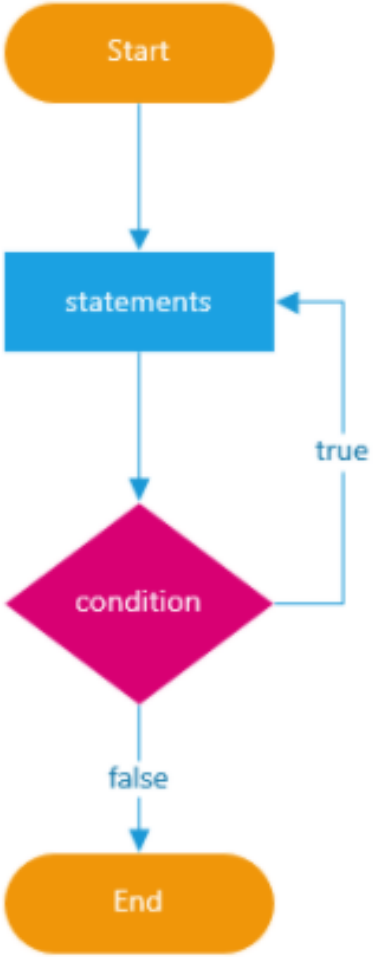
```
do {
  statement(s);
} while(expression);
```

Unlike the `while` loop, the `do-while` loop always executes the body at least once before it evaluates the expression.

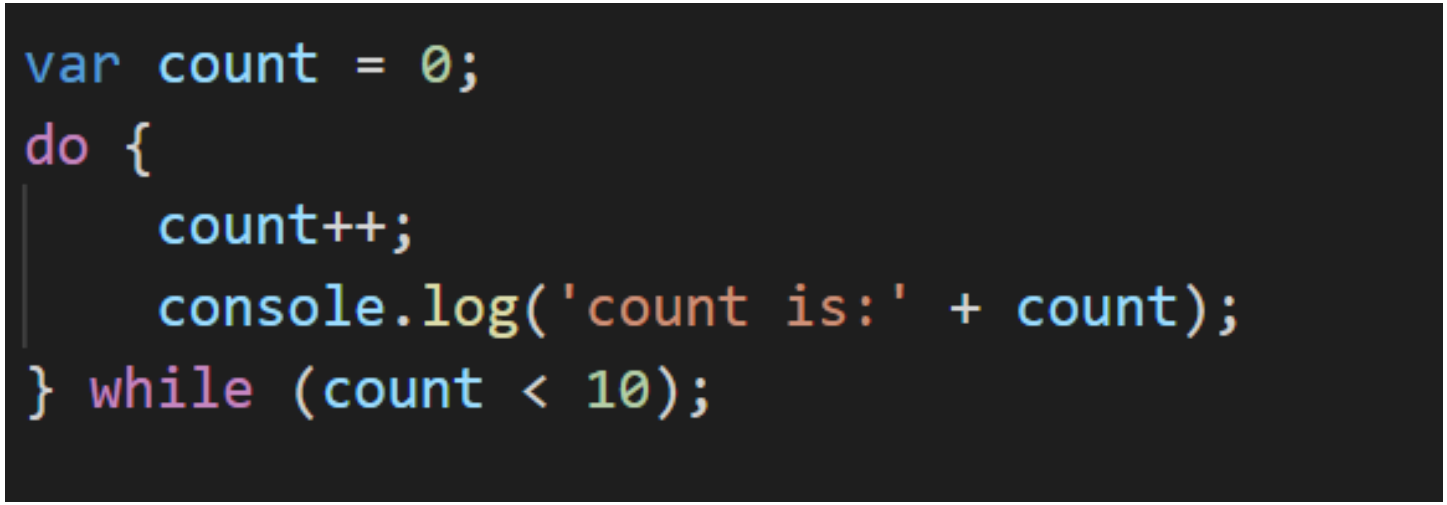
Because the expression is evaluated only after the body of the loop has been executed, the `do-while` loop is called a post-test loop.

Inside the body of the loop, you need to make changes to some variable to ensure that the expression evaluates to `false` after iterations. Otherwise, you will have an indefinite loop.

Program flow (do-while Loop)



Program flow (do-while Loop)



Functions

- When you write a program, you often need to **perform the same action in many places.**
- For example, you want to show a message to the users when they complete an action.
- **To avoid repeating** the same code all over places, you can use a function to wrap that code and reuse it.
- JavaScript provides many built-in functions such as alert() and console.log(). In this tutorial, you will learn how to develop custom functions.

Functions(cont.)

Declaring functions

To declare a function, you use the function keyword, followed by the function name, a list of parameters, and the function body as follows:

```
function functionName(parameters) {
    // function body
    // ...
}
```

- The function name must be a valid JavaScript **identifier**.
- By convention, the function name should start with a verb like **getData(), fetchContents(), or isValid().**
- A function can accept zero, one, or multiple **parameters**. If there are multiple parameters, you need to separate them by commas (,).

Functions(cont.)

JavaScript

The following declares a function named `say()` that accepts no parameter:

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

The following declares a function named `square()` that accepts one parameter:

```
function square(a) {
    //...
}
```

And the following declares a function named `add()` that accepts two parameters:

```
function add(a, b) {
    // ...
}
```

Functions(cont.)

- **Calling functions**

Calling functions

To call a function, you use its name followed by the function arguments enclosed in parentheses, like this:

```
functionName(arguments);
```

When you call a function, the function executes the code inside its body. This process is also known as invocation. In other words, you call a function or invoke a function to execute it.

For example, the following shows how to call the `say()` function:

```
say('Hello');
```

The `'Hello'` string is an argument that we pass into the `say()` function.

Functions(cont.)

- **Returning a value**
-

```
function say(message) {
    console.log(message);
}

var result = say('Hello');
console.log('Result:', result);
```

Output:

```
Hello
Result: undefined
```

To specify a return value for a function, you use the the `return` statement followed by an expression or a value, like this:

```
return expression;
```

Functions(cont.)

- **Function hoisting**

In JavaScript, it is possible to use a function before it is declared. See the following example:

```
showMe(); // a hoisting example

function showMe(){
    console.log('an hoisting example');
}
```

This feature is called hoisting.

The function hoisting is a mechanism that the JavaScript engine physically moves function declarations to the top of the code before executing them.

The following shows the version of the code before the JavaScript engine executes it:

```
function showMe(){
    console.log('a hoisting example');
}

showMe(); // a hoisting example
```

Functions(cont.)

JavaScript

- **Anonymous Functions**

An anonymous function is a function without a name. An anonymous function is often not accessible after its initial creation.

The following shows an anonymous function that displays a message:

```
let show = function () {
    console.log('Anonymous function');
};

show();
```

In this example, the anonymous function has no name between the `function` keyword and parentheses `()`.

Because we need to call the anonymous function later, we assign the function to the `show` variable.

Functions(cont.)

- **Self invoking function (**Immediately Invoked Function)
- A JavaScript immediately invoked function expression is a function defined as an expression and executed immediately after creation. The following shows the syntax of defining an immediately invoked function expression:

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

Why IIFEs

When you define a function, the JavaScript engine adds the function to the global object. See the following example:

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

On the Web Browsers, the `add()` function is added to the `window` object:

```
console.log(window.add);
```

Functions(cont.)

- **Self invoking function (**Immediately Invoked Function)

Similarly, if you declare a variable outside of a function, the JavaScript engine also adds the variable to the global object:

```
var counter = 10;
console.log(window.counter); // 10
```

If you have many global variables and functions, the JavaScript engine will only release the memory allocated for them until when the global object loses the scope.

As a result, the script may use the memory inefficiently. On top of that, having global variables and functions will likely cause the name collisions.

One way to prevent the functions and variables from polluting the global object is to use immediately invoked function expressions.

Functions(cont.)

- **Self invoking function (**Immediately Invoked Function)

In JavaScript, you can have the following expressions:

```
'This is a string';
(10+20);
```

This syntax is correct even though the expressions have no effect. A function can be also declared as an expression which is called a function expression:

```
let add = function(a, b) {
    return a + b;
}
```

Functions(cont.)

- **Self invoking function (**Immediately Invoked Function)

The following expression is called an immediately invoked function expression (IIFE) because the function is created as an expression and executed immediately:

```
(function(a,b){
        return a + b;
})(10,20);
```

This is the general syntax for defining an IIFE:

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

Note that you can use an arrow function to define an IIFE:

```
(() => {
    //...
})();
```

Objects

JavaScript

In JavaScript, an object is a collection of properties, defined as a key-value pair. Each property has a key and a value. The property key can be a string and the property value can be any valid value.

To create an object, you use the object literal syntax. For example, the following snippet creates an empty object:

```
let empty = {};
```

To create an object with properties, you use the `key : value` syntax. For example, the following snippet creates a `person` object:

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

The `person` object has two properties `firstName` and `lastName` with the corresponding values `'John'` and `'Doe'`.

Objects (cont.)

Accessing properties

To access a property of an object, you use one of two notations: the dot notation and array-like notation.

1) The dot notation (.)

The following illustrates how to use the dot notation to access a property of an object:

```
objectName.propertyName
```

For example, to access the `firstName` property of the `person` object, you use the following expression:

```
person.firstName
```

The following snippet creates a `person` object and shows the first name and last name on the Console:

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};

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

Objects (cont.)

2) Array-like notation ([ ])

The following illustrates how to access the value of an object's property via the array-like notation:

```
objectName['propertyName'];
```

For example:

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(person['firstName']);
console.log(person['lastName']);
```

Objects (cont.)

Add a new property to an object

Unlike objects in other programming languages such as Java and C#, you can add a property to an object after creating it.

The following statement adds the `age` property to the `person` object and assigns 25 to it:

```
person.age = 25;
```

Delete a property of an object

To delete a property from an object, you use the `delete` operator:

```
delete objectName.propertyName;
```

The following example removes the `age` property from the `person` object:

```
delete person.age;
```

Objects (cont.)

Check if a property exists

To check if a property exists in an object, you use the in operator:

```
propertyName in objectName
```

The following example creates an `employee` object and uses the `in` operator to check if the `ssn` and `employeeId` properties exist in the object.

```
let employee = {
    firstName: 'Peter',
    lastName: 'Doe',
    employeeId: 1
};

console.log('ssn' in employee);
console.log('employeeId' in employee);
```

Objects (cont.)

Iterate over properties of an object using for...in loop

To iterate over all properties of an object without knowing property names, you use the `for...in` loop:

```
for(let key in object) {
    // ...
};
```

For example, the following statement creates a `website` object and iterates over its properties using the `for...in` loop:

```
let website = {
    title: 'JavaScript Tutorial',
    url: 'https://www.javascripttutorial.net',
    tags: ['es6', 'javascript', 'node.js']
};

for (const key in website) {
    console.log(website[key]);
}
```

JavaScript Events

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

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

HTML 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

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:

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

JavaScript Events (cont.)

With single quotes:

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

With double quotes:

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

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

Example

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

JavaScript Events (cont.)

JavaScript

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

Example

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

JavaScript Events (cont.)

Event handler	Description
onMouseDown	when pressing any of the mouse buttons.
onMouseMove	when the user moves the mouse pointer within an element.
onMouseOut	when moving the mouse pointer out of an element.
onMouseUp	when the user releases any mouse button pressed
onMouseOver	when the user moves the mouse pointer over an element.
onClick	when clicking the left mouse button on an element.
onDblClick	when Double-clicking the left mouse button on an element.
onDragStart	When the user has begun to select an element

JavaScript Events (cont.)

Event handler	Description
onKeyDown	When User presses a key
onKeyPress	When User holds down a key
onKeyUp	When User a key

JavaScript Events (cont.)

Event handler	Description
onAbort	The User interrupted the transfer of an image
onBlur	when loosing focus
onFocus	when setting focus
onChange	when the element has lost the focus and the content of the element has changed
onLoad	a document or other external element has completed downloading all the data into the browser
onUnload	a document is about to be unloaded from the window
onError	When an error has occurred in a script.
onMove	when moving the browser window

JavaScript Events (cont.)

Event handler	Description
OnReset	When the user clicks the form reset button
onSubmit	When the user clicks the form submit button
onScroll	When the user adjusts an element's scrollbar
onResize	When the user resizes a browser window
onHelp	When the user presses the F1 key
onselect	When selecting text in an input or a textarea element
onStart	When A marquee element loop begins
onFinish	When a marquee object finishes looping
onSelectStart	When the user is beginning to select an element

JavaScript Strings

A JavaScript string is zero or more characters written inside quotes.

Example

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

Try it Yourself »

You can use single or double quotes:

Example

```
var carName1 = "Volvo XC60";  // Double quotes
var carName2 = 'Volvo XC60';  // Single quotes
```

JavaScript Strings

The length of a string is found with the built-in `length` property :

Example

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

JavaScript Strings

Finding a String in a String

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

Example

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

JavaScript Strings

JavaScript

JavaScript counts positions from zero.

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

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

Example

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

JavaScript Strings

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

Example

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

JavaScript Strings

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

Example

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

JavaScript Strings

Extracting String Parts

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

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

JavaScript Strings

The slice() Method

`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):

Example

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

The result of res will be:

```
Banana
```

JavaScript Strings

> Remember: JavaScript counts positions from zero. First position is 0.

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:

Example

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

The result of res will be:

```
Banana
```

JavaScript Strings

The substring() Method

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

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

Example

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

The result of *res* will be:

```
Banana
```

JavaScript Strings

The substr() Method

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

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

Example

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

The result of res will be:

```
Banana
```

JavaScript Strings

Replacing String Content

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

Example

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

JavaScript Strings

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

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

Example

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

JavaScript Strings

Converting to Upper and Lower Case

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

Example

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

JavaScript Strings

The concat() Method

`concat()` joins two or more strings:

Example

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

JavaScript Strings

Extracting String Characters

There are 3 methods for extracting string characters:

- `charAt(position)`
- `charCodeAt(position)`
- Property access [ ]

The charAt() Method

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

Example

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

JavaScript Dialog boxes

JavaScript

Prompt:

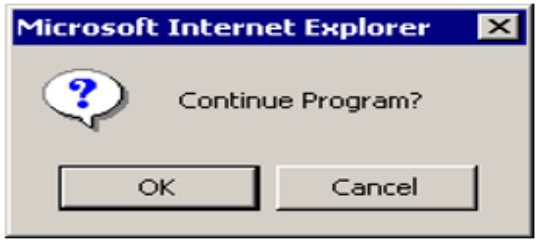
- The simplest way to interact with the user.

- Example:

```
<script>
        var Name = prompt('User Name' , 'Enter your name');
</script>
```

JavaScript Dialog boxes(cont.)

- **Confirm:**
  - displays a dialog box with two buttons: OK and Cancel.
    - If the user clicks on OK it will return true.
    - If the user clicks on the Cancel it will return false.



  - Example:

```
<script>
        var response = confirm('Are you sure you want to continue?');
</script>
```

JavaScript built in functions

JavaScript Built-in Functions

Name	Description	Example

parseInt()	Convert string to int	parseInt("3") //returns 3
		parseInt("3a") //returns 3
		parseInt("a3") //returns NaN

parseFloat()	Convert string to float	parseFloat("3.55") //returns 3.55
		parseFloat("3.55a") //returns 3.55
		parseFloat("a3.55") //returns NaN

Number()	The Number() function converts the object argument to a number that represents the object's value.
	
	if the value cannot be converted to a legal number, NaN is returned .
		var x1 = false, x2 = "999", x3 = "999 888";
		document.write(Number(x1), Number(x2), Number(x3));
		// returns 0, 999, Nan
		Note:
		parseInt("123hui"); //returns 123
		Number("123hui"); //returns NaN

String()	Convert different objects to strings.	var x1 = New Boolean(0);, x2 = 999, x3 = "999 888";
		document.write(String(x1), String(x2), String(Sx3));
		// returns false, 999, 999 888

JavaScript built in functions

Name	Description	Example

isFinite(num)
(used to test number)
	returns true if the string contains numbers only, else false	document.write(isFinite(33))
		//returns true
		document.write(isFinite("Hello"))
		//returns false
		document.write(isFinite("33a"))
		//returns false

isNaN(val)
(used to test string)
	validate the argument for a number and returns true if the given value is not a number else returns false.	document.write(isNaN(0/0)) //returns true
		document.write(isNaN("348a")) //returns true
		document.write(isNaN("abc")) //returns true
		document.write(isNaN("348")) //returns false

eval(expression)	evaluates an expression and returns the result.	a=999; b=777;
		document.write(eval(b + a));
		// returns 1776

JavaScript built in functions

Name	Description	Example
escape(string)	method converts the special characters like space, colon etc. of the given string in to escape sequences.	escape("test val"); //test%20val
unescape(string)	function replaces the escape sequences with original values. e.g. %20 is the escape sequence for space " ".	unescape("test%20val"); //test val