

Set 18: Writing Programs in JavaScript

Skill 18.01: Describe the parts of a simple computer program

Skill 18.02: Interpret AP CSP pseudocode

Skill 18.03: Reference a JavaScript file from an HTML

Skill 18.04: Describe the purpose of the *defer* attribute

Skill 18.05: Print to the console with JavaScript

Skill 18.06: Add comments to a JavaScript program

Skill 18.01: Describe the parts of a simple computer program

Skill 18.01 Concepts

A computer program is a set of instructions we want a computer to perform. Just like telling a dog to “sit” or “beg”, we can tell a computer to “add” or “print”.

Consider a simple “Hello World” program, the first program of many new programmers. To get our computer to print “Hello World” to the screen we could write something like the following,

Code	Output
1 <code>println("Hello World!")</code>	Hello World

Let’s break down our simple program above to better understand how it works.

<code>println("Hello World!")</code>	This line of code is called a statement . All programs are made up of statements, and each statement is an instruction to the computer about something we need it to do.
1	The number 1 in front of the statement is called a line number. This is very important for referring to different parts of a program and for debugging.
<code>println</code>	The <code>println</code> command is also called a function, method, or procedure . It tells the computer to call the procedure named <code>println</code> and output something to the screen.
<code>("Hello World!")</code>	The information in parentheses following the <code>println</code> command is called a parameter . A parameter specifies the information we want to provide to the function. If nothing is provided, nothing will be printed to the screen. For example, <code>println()</code> , would print nothing

Display commands are different in each environment and language. The `println()` command described here is not actual code, but is referred to as pseudocode.

[Skill 18.01 Exercise 1](#)

Skill 18.02: Interpret AP CSP pseudocode

Skill 18.01 Concepts

The programming language we will be using in this class is JavaScript. There are a large number of programming languages out there, and you may have already used a few others like Code.org, Scratch, or Python.

Thankfully, we have another way to describe programs: pseudocode. Pseudocode is a language that doesn't actually run anywhere, but still represents programming concepts that are common across programming languages. There are different flavors of pseudocode, so here we'll use the one that's used by the AP CSP exam.

Below is AP CSP pseudocode for displaying output,

```
DISPLAY (expression)
```

That line of pseudocode means "displays the value of expression followed by a space."

For example, you might see pseudocode like this:

```
DISPLAY ("Howdy")  
DISPLAY ("Neighbor!")
```

The above represents code that outputs: "Howdy Neighbor! "

The `DISPLAY (expression)` pseudocode is similar to the `println(expression)` that we just learned, but they're slightly different because `DISPLAY()` represents code that adds a space while `println()` adds a new line.

As we go through and learn programming concepts, you will also be shown the pseudocode for each of them.

Skill 18.01: Reference a JavaScript file from an HTML page

Skill 18.01 Concepts

Unlike pseudocode, all modern browsers are capable of rendering JavaScript, which makes developing in JavaScript quick and easy. Rendering JavaScript however requires that we *tell* the browser where the script is located. This done in the head section of the HTML page. Below is an illustration of what this looks like.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width">
  <script src = "app.js"></script>
  <title>My First JavaScript Program</title>
</head>
<body>

</body>
</html>
```

Telling the browser where your JavaScript code is located is done in the head section of your HTML page.

Including JavaScript on an HTML page requires the `<script></script>` tag. It is best practice to keep your JavaScript code separate from your HTML code. That is, all JavaScript code should be written and saved in a separate JavaScript file. Referencing a JavaScript file requires the `src` attribute. `src` in this case stands for “source”. Below is an illustration of how this works,

MyFirstProgram

JS app.js

<> index.html

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <script src = "app.js"></script>
  <title>My First JavaScript Program</title>
</head>
<body>

</body>
</html>
```

app.js

```
console.log("Hello!");
```

When the browser reaches this line in the HTML page, the code in the app.js file runs

The example below illustrates how to include a JavaScript file from an HTML page. In the example, the App.js file is located in the same directory as the Index.html file.

MyWebsite	
index.html App.js	<script src = "App.js"></script>

Now consider an example where the *App.js* file we are trying to reference is stored in a directory that is different than *Index.html*. In the file structure below, we have created a directory called *Scripts* and placed the *App.js* file inside it. The following code could be used to reference the *App.js* file from the *Index.html* page.

MyWebsite	<code><script src = "Scripts/App.js"></script></code>
index.html	
Scripts	
App.js	

Finally consider the situation below. *Index.html* and *App.js* are both in separate directories in the *MyWebsite* directory. Inside the *Home* directory we have an *Index.html* page and inside the *Scripts* directory we have our *App.js* file we want to reference. To do this, we must first “backout” of the *Home* directory, then enter the *Scripts* directory. The “.” syntax is used to backout of a directory.

MyWebsite		<code><script src = "../Scripts/App.js"></script></code>
Home	Scripts	
Index.html	App.js	

[Skill 18.03 Exercise 1](#)

Skill 18.04: Describe the purpose of the *defer* attribute

Skill 18.04 Concepts

In modern websites, scripts are often “heavier” than HTML: their download size is larger, and processing time is also longer.

When the browser loads HTML and comes across a `<script></script>` tag, it can’t continue building the HTML page until the script is finished executing. The same happens for external scripts a `<script src=“...”></script>` - The browser must wait for the script to download, execute the downloaded script, and only then can it process the rest of the page.

That leads to two important issues:

1. Scripts can’t see HTML elements below them, so they can’t manipulate them
2. If there’s a bulky script at the top of the page, it “blocks the page”. Users can’t see the page content until the script downloads and runs. This is illustrated below.

```
<!DOCTYPE html>
<html>
<head>
<script src="Scripts/App.js"></script>
</head>
<!-- This isn't visible until the script above loads and runs -->
<body>
  <p>...content after script...</p>
</body>
</html>
```

There are some workarounds for this issue. For instance, we can put a script at the bottom of the page. Then it can see elements above it and does not block the page content from displaying.

```
<!DOCTYPE html>
<html>
<head></head>
<body>
  ...all content is above the script...

<script src="Scripts/App.js"></script>
</body>
</html>
```

But this solution is far from perfect. For example, the browser cannot see the script (and cannot start downloading it) until the full HTML document has loaded. For long HTML documents, that may be a noticeable delay.

Such things are invisible for people using very fast connections, but many people in the world still have slow internet speeds and use a far-from-perfect mobile internet connection.

Luckily, there is a solution that solves this issue.

The `defer` attribute tells the browser not to wait for the script. Instead, the browser will continue to process the HTML. The script loads “in the background”, and then runs when the document is fully built.

Here's the same example as above, but with defer.

```
<!doctype html>
<html>
<head>
<script defer src="Script/App.js"></script>
</head>
<body>
<p>...content before script...</p>

<!-- visible immediately -->
<p>...content after script...</p>
</body>
</html>
```

[Skill 18.04 Exercise 1](#)

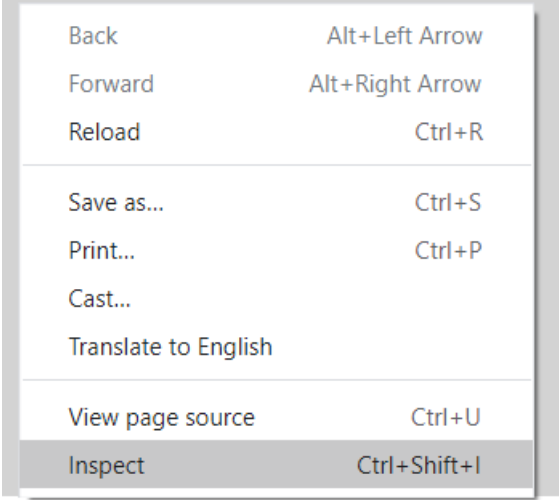
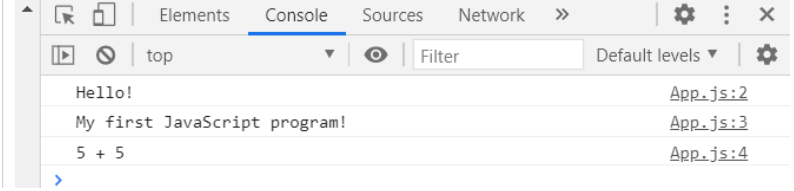
Skill 18.05 Print to the console with JavaScript

Skill 18.05 Concepts

All JavaScript files must end with the *.js extension. The extension communicates to the browser to execute the code inside the file.

All browsers have consoles that can display important messages, like errors, for developers. Much of the work the computer does with our code is invisible to us by default. If we want to see things that do not appear on our screen, we can print, or log, to the console directly. Using the console to log messages is extremely useful for debugging and interpreting the programs we write.

To view the console from your browser do the following type *Ctrl-Shift-J*, or,

<p>Right click on the webpage where your JavaScript is running and select <i>inspect</i>.</p>	
<p>Select the Console tab</p>	

In JavaScript, the `console` keyword refers to an object, a collection of data and actions, that we can use in our code. Keywords are words that are built into the JavaScript language, so the computer will recognize them and treats them specially.

One action, or method, that is built into the console object is the `.log()` method. When we write `console.log()`; what we put inside the parentheses will get printed, or logged, to the console.

3 `console.log(5);`

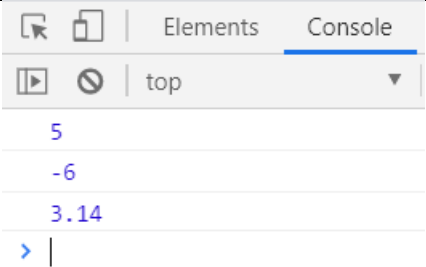
the value printed (pointing to 5)

denotes the end of the line (pointing to the semicolon)

The example above prints the number 5 to the console. The semicolon denotes the end of the line, or statement.

Printing numbers to the console

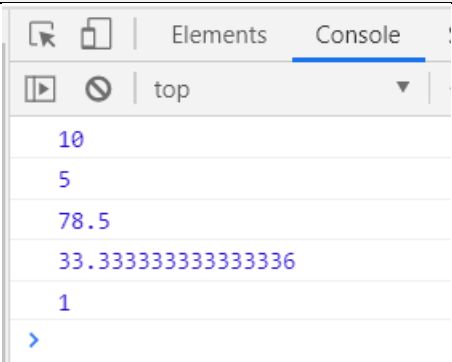
If you want to print numbers to the console, you can do so by indicating the number you want to print in parentheses. This is illustrated in the example above. In addition to whole numbers you can also print negative numbers and decimals,

Code	Output
<pre>console.log(5); console.log(-6); console.log(3.14);</pre>	

In addition to printing single numbers, you can print the result of simple arithmetic operations. The symbols used for carrying out simple arithmetic operations are as follows,

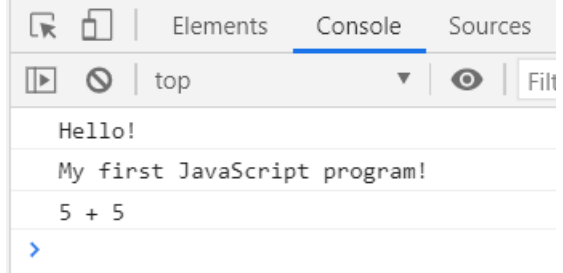
Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
%	modulus

Below are a few examples,

Code	Output
<pre>console.log(5 + 5); console.log(10 - 5); console.log(3.14*5*5); console.log(100/3); console.log(21%2);</pre>	

Printing text to the console

Text can also be printed to the console by enclosing the text you want printed in quotes. Notice in the last example, `5 + 5` is printed to the console instead of 10 because it is in quotes.

Code	Output
<pre>console.log("Hello!"); console.log("My first JavaScript program!"); console.log("5 + 5");</pre>	

[Skill 18.05 Exercise 1](#)

Skill 18.06 Add comments to a JavaScript program

Skill 18.06 Concepts

As we write JavaScript, we can write comments in our code that the computer will ignore as our program runs. These comments exist just for human readers.

Comments can explain what the code is doing, leave instructions for developers using the code, or add any other useful annotations.

There are two types of code comments in JavaScript: single line and multiline

- A single line comment will comment out a single line and is denoted with two forward slashes `//` preceding it.

```
// Prints 5 to the console
console.log(5);
```

- You can also use a single line comment to comment after a line of code:

```
console.log(5); // Prints 5
```

- A multi-line comment will comment out multiple lines and is denoted with `/*` to begin the comment, and `*/` to end the comment.

```
/*  
This is all commented  
console.log(10);  
None of this is going to run!  
console.log(99);  
*/
```

- You can also use this syntax to comment something out in the middle of a line of code:

```
console.log(/*IGNORED!*/ 5); // Still just prints 5
```

[Skill 18.06 Exercise 1](#)