JavaScript

Morteza Ghodoousi

Your First Lesson

Have you ever visited a website that made you think…"Wow, this website is really cool and interactive"?

Well, JavaScript was probably making it happen.

In this course, you'll learn to code with JavaScript -

one of the most popular programming languages that makes websites dynamic and interactive.

You can also use JavaScript to create mobile apps and games, process data, and much more!

Your First Lesson

JavaScript is a programming language that runs in web browsers.

Most websites use JavaScript and when things go wrong, web developers use the console to investigate errors.

In this lesson you'll learn about the web developer’s main weapon: the developer’s console.

Coding errors are not visible in the browser. Web developers use the console to test code and fix bugs.

Your First Lesson

The console is part of the web browser.

Logging (writing) messages to the console is a good way to diagnose and troubleshoot minor issues in your code.

You can use console.log() to write or log messages.

In the example code below, the message 'All good, no errors' is sent to the console.

```
console.log('All good, no errors')
```

You can log messages to inform what the code is doing or alert that there’s an issue.

Text in JavaScript needs to be enclosed in quotes

Your JavaScript code won’t work if you don’t include a pair of quotation marks around the text message.

You can use either single or double quotation marks.

Your First Lesson

You are fantastic! In this lesson, you've learned that:

- JavaScript is a very popular programming language.
- Web developers use the console to test code and fix bugs.
- You can use console.log() to write or log messages to the console.

Output

Let's use JavaScript to print "Hello World" to the browser. This is what that would look like.

document.write("Hello World!");

Notice some extra stuff there? Nothing gets past you!

Time to introduce the document.write() function. This is what we need to use to write text into our HTML document.

You can also use standard HTML markup language to customize the appearance text in the output:

document.write("<h1>Hello World!</h1>");

Heads up!

document.write() should be used only for testing. We'll cover some other output mechanisms real soon.

Output

Right, we're now experts in writing HTML output with document.write().

Time for a different type of output. Let's learn about output to the browser console.

For this we'll be needing the trusty console.log() function.

The console is part of the web browser and allows you to log messages, run JavaScript code, and see errors and warnings.

It looks like this:

```
console.log('All good, no errors')
```

Heads Up!

Devs mostly use the console to test their JavaScript code.

Output

Right, we're now experts in writing HTML output with document.write().

Time for a different type of output. Let's learn about output to the browser console.

For this we'll be needing the trusty console.log() function.

The console is part of the web browser and allows you to log messages, run JavaScript code, and see errors and warnings.

It looks like this:

```
console.log('All good, no errors')
```

Heads Up!

Devs mostly use the console to test their JavaScript code.

Task

Your First Program

Write a program to print "JS is fun".
Note that the sentence starts with **capital letters**.

Hint
Use **console.log()** function.

Remember to enclose the text into **double quotes**.

Variables

Variables are containers for storing data values. The value of a variable can change throughout the program.

Declaring a variable is as simple as using the keyword var. Which would look like this:

var x = 10;

In this example we've assigned a value of 10 to the variable x.

We've used the word assigned deliberately here, because in JavaScript,

the equal sign (=) is actually called the "assignment" operator, rather than an "equal to" operator.

Which means that in JavaScript, x = y will assign the value of y to x variable.

Heads up!

JavaScript is sensitive, case sensitive that is. So variables like lastName and lastname are not the same.

Variables

Ok, let’s put some of what we’ve learned together!

How about we assign a value to a variable and output it to the browser. We’ve got this!

```
var x = 100;

document.write(x);
```

But what’s the point of variables anyway?

Well, imagine your program has 1000 lines of code that include the variable x. With variables you can change the value of the variables and use them multiple times in your code: Like this:

```
var x = 100;

document.write(x);

x = 42;

document.write(x);
```

Heads Up!

Every written "instruction" is called a statement. JavaScript statements are separated by semicolons.

Variables

Let's talk about names.

It's super important to remember that JavaScript variable names are case-sensitive.

Brace yourself for more rules!

- The first character of a variable name must be a letter, underscore (_), or a dollar sign ($) (Subsequent characters can be letters, digits, underscores, or dollar signs).
- The first character of a variable name can't be a number.
- Variable names can't include a mathematical or logical operator in their name. For instance, 2*something or this+that;
- Variable names can't contain spaces.
- You're not allowed to use any special symbols, like my#num, num%, etc.

Heads up!

JavaScript is a hyphen free zone. They're reserved for subtractions.

Comments

Ok, let’s talk about comments in JavaScript.

So we know about statements, these are the instructions within our program that get "executed" when the program runs.

But! Not all JavaScript statements are "executed".

Any code after a double slash //, or between /* and */, is treated as a comment, and will be ignored, and not executed.

But why write code that is never going to be executed. Isn’t that a waste of time?

Not at all! Comments are a good idea, especially ones relating to large functions,

as they help make our code more readable for others. So be kind, and comment!

Heads up!

alert() is used to create a message box.

Heads up!

We use comments to describe and explain what the code is doing.

Task

Comments

You are given a program that prints **two expressions**.

Task
Comment the second line of the code to print only the first expression.

Use **double slashes (//)** before the code line in order to comment it out.

Data Types

The term data type refers to the types of values a program can work with.

The sky's the limit with JavaScript variables, which can hold a bunch of different data types–numbers, strings, arrays, you name it.

Let's start simple though.

Numbers can be written with or without decimals. Like this:

```
var num = 42; // A number without decimals

var price = 55.55;

document.write(price);
```

Heads up!

Changing this variable is a breeze, just assign to it any other data type value, like num = 'some random string'.

Data Types

In JavaScript we can use strings to store and manipulate text.

A string can be any text wrapped in quotes. Single or double quotes, it doesn’t matter, so long as you’re consistent with them. Like this:

```
var name = 'John';
var text = "My name is John Smith";
```

What if we want to use quotes inside a string though?? No problem!

You can use quotes inside a string, as long as they don't match the quotes enclosing the string itself. Take a look:

```
var text = "My name is 'John' ";
```

Heads up!

You can get double quotes inside of double quotes using the escape character like this: \" or \' inside of single quotes.

Data Types

Now is a good time to talk about the backslash (\) escape character.

It comes to the rescue when you need to put quotes within strings

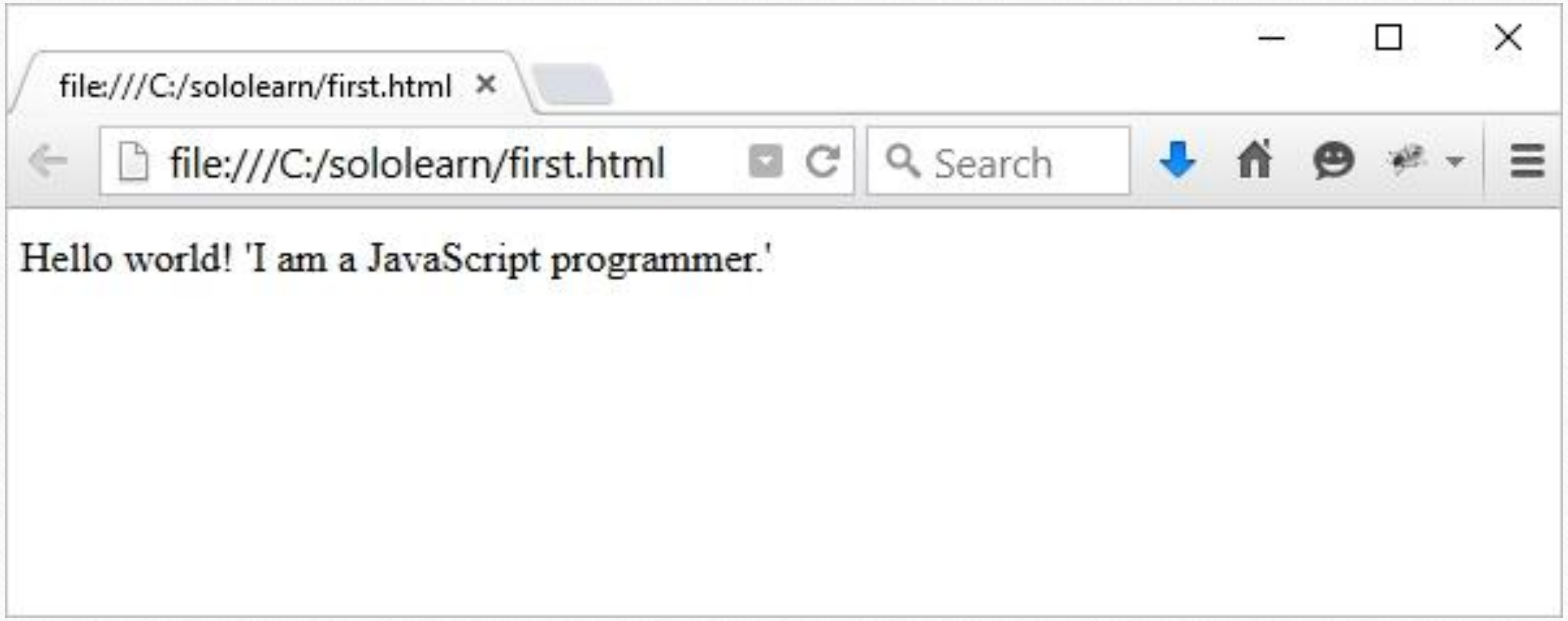
(and a bunch of other situations) by transforming special characters into string characters.

Take a look:

```
var sayHello = 'Hello world! \'I am a JavaScript programmer.\' ';
document.write(sayHello);
```

Data Types

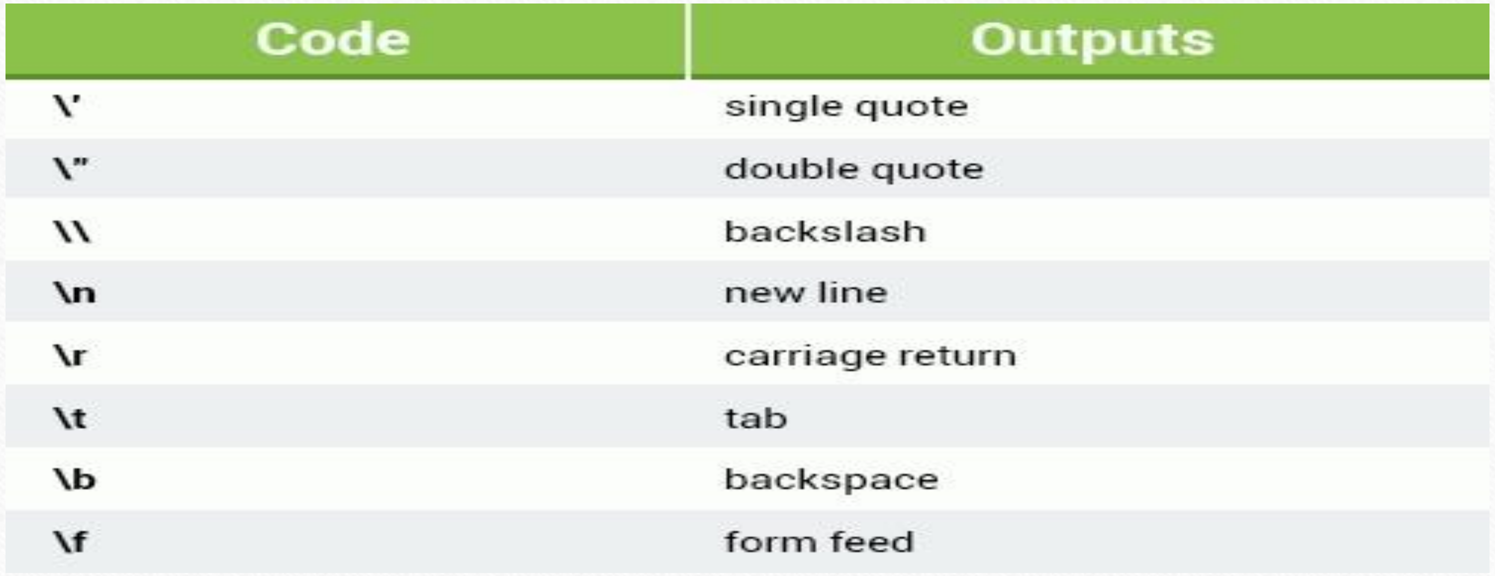
Result:


file:///C:/sololearn/first.html
file:///C:/sololearn/first.html
Search

```
Hello world! 'I am a JavaScript programmer.'
```

Data Types

But the escape character (\) isn’t just for quotes, it works when you need to put other special characters inside strings too!



Code	Outputs
\'	single quote
\"	double quote
\\	backslash
\n	new line
\r	carriage return
\t	tab
\b	backspace
\f	form feed

Data Types

Heads up!

If you start a string with a single quote, then you need to end it with a single quote too.

This applies to double quotes. Otherwise, JavaScript will get confused.

Data Types

Not just fun to say, Booleans in JavaScript serve a useful function by letting you have one of two values, either true or false.

So when you need a data type that can only have one of two possible values, like Yes/No, on/off or true/false, look no further than Mr Boolean.

Let's look at an example:

var isActive = true;

var isHoliday = false;

Heads up!

The Boolean value of 0 (zero), null, undefined, empty string is false.

Everything with a "real" value is true.

Task

Escaping Characters in Strings

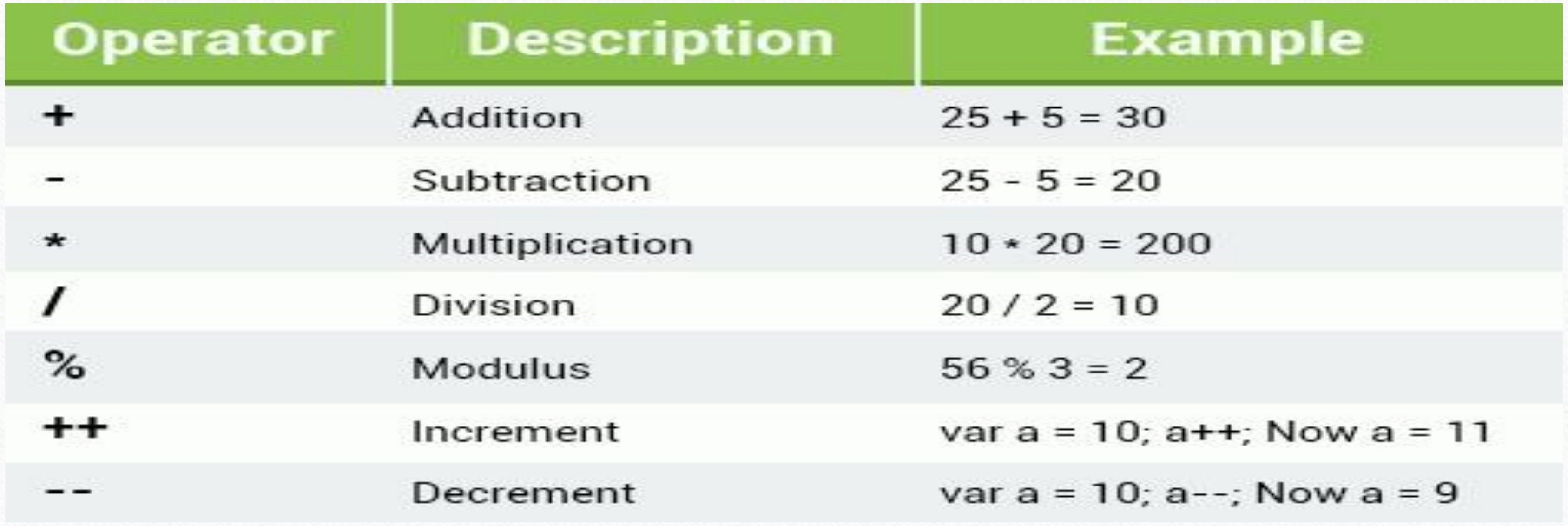
Fix the given code to output the following expression:
I'm learning JavaScript

Hint
Escape the single quote **inside the string**.

Use **backslash \ ** for escaping.

Math Operators

The name might be a bit of a giveaway but, Arithmetic operators pretty much perform arithmetic functions on numbers (both literals and variables).



Operator	Description	Example
+	Addition	25 + 5 = 30
-	Subtraction	25 - 5 = 20
*	Multiplication	10 * 20 = 200
/	Division	20 / 2 = 10
%	Modulus	56 % 3 = 2
++	Increment	var a = 10; a++; Now a = 11
--	Decrement	var a = 10; a--; Now a = 9

Math Operators

Below you can see the addition operator (+) in action determining the sum of two numbers.

```
var x = 10 + 5;
document.write(x);
```

You can add as many numbers or variables together as you want or need to.

```
var x = 10;
var y = x + 5 + 22 + 45 + 6548;
document.write(y);
```

Heads up!

You can get the result of a string expression using the eval() function,

which takes a string expression argument like eval("10 * 20 + 8") and returns the result.

If the argument is empty, it returns undefined.

Math Operators

What tool is best suited for math?...Multi-pliers!

JavaScript is pretty good at math too though!

We use the * operator to multiply one number by the other.

Heads up!

10 * '5' or '10' * '5' will give the same result.

But trying to multiply a number with string values that aren't numbers, like 'Hello World!' * 5 will return NaN (Not a Number).

Math Operators

We use the / operator to perform division operations.

Heads up!

Beware of situations where there could be a division by 0, things get messed up when we do impossible math!

The Modulus (%) operator returns the division remainder (what's left over).

Heads up!

In JavaScript, we can use the modulus operator on integers AND on floating point numbers.

Math Operators

- Increment ++

The increment operator increases the numeric value of its operand by 1.

When placed before the operand, it'll return the incremented value.

When placed after it, it'll return the original value and then increments the operand.

- Decrement --

The decrement operator decreases the numeric value of its operand by 1.

When placed before the operand, it'll return the decremented value. When placed after the operand,

it'll return the original value and then decrements the operand.

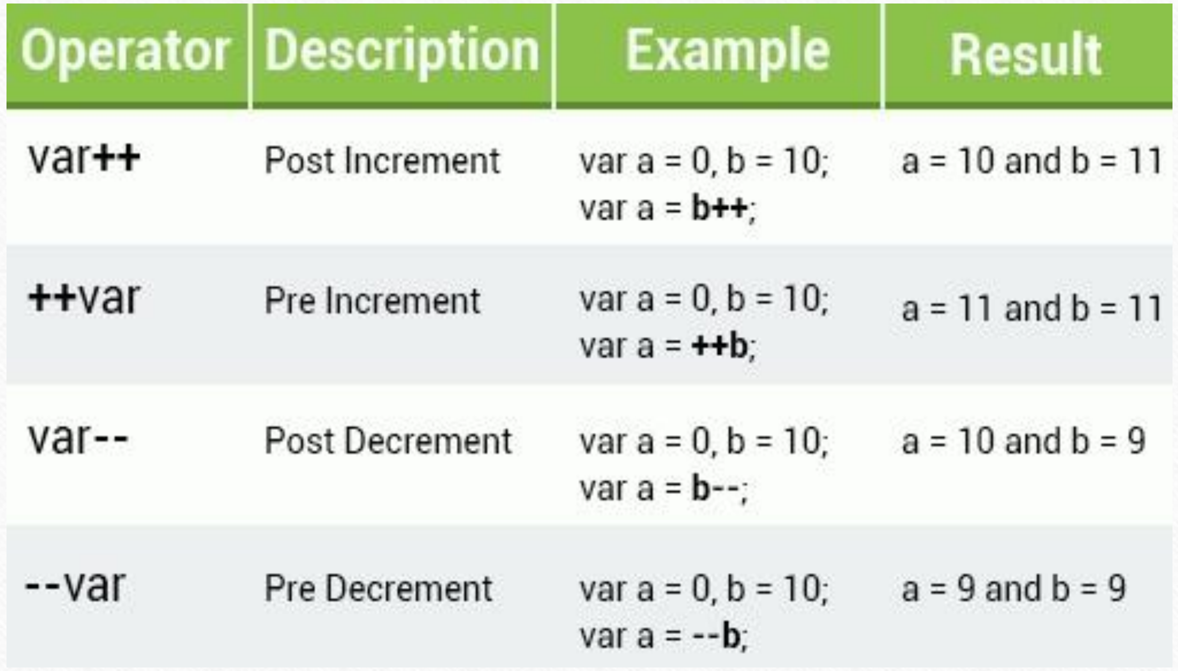
Math Operators

Some examples:

Heads up!

Just like the math you learned in school, you can change the order of the arithmetic operations by using parentheses.

Like this: var x = (100 + 50) * 3;



Operator	Description	Example	Result
var++	Post Increment	var a = 0, b = 10; var a = **b++**;	a = 10 and b = 11
++var	Pre Increment	var a = 0, b = 10; var a = **++b**;	a = 11 and b = 11
var--	Post Decrement	var a = 0, b = 10; var a = **b--**;	a = 10 and b = 9
--var	Pre Decrement	var a = 0, b = 10; var a = **--b**;	a = 9 and b = 9

Task

Math Operators

In the office, 2 monitors are connected to each computer.
The first line of the given code takes the **number of computers** as input.
Task
Complete the code to calculate and output the number of monitors to the console.

Sample Input
10
Sample Output
20
Hint
Since each computer has 2 monitors, simply multiply the count of the computers by 2.

Use the **multiplication operator (*)**.

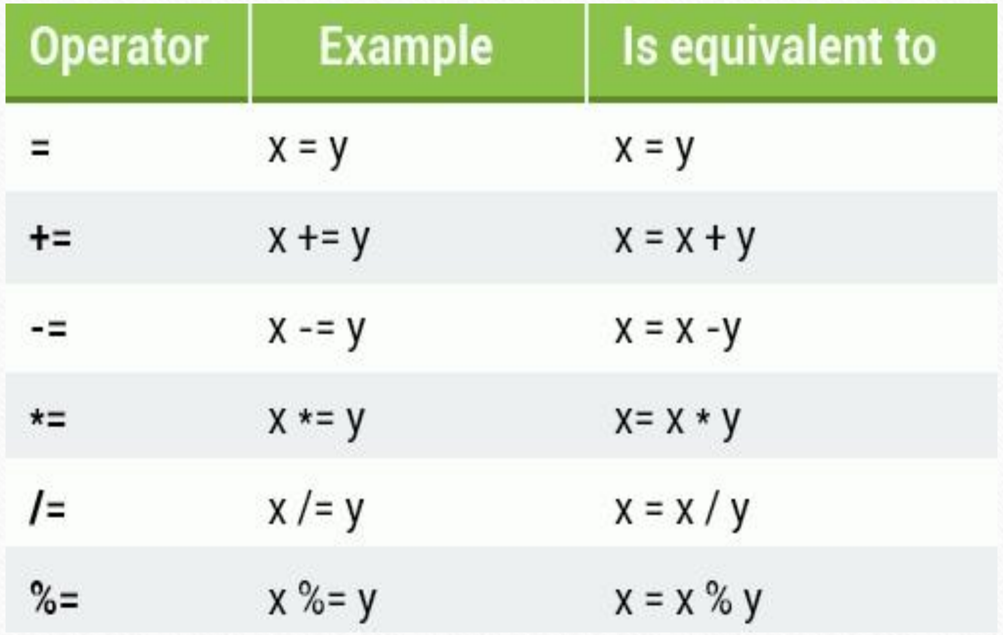
Assignment Operators

Next in a series very logically named operators is Assignment operators!

And you guessed it, we use these guys to assign values to JavaScript variables.

Heads up!

You can use multiple assignment operators in one line, such as x -= y += 9.



Operator	Example	Is equivalent to
=	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

Comparison Operators

We can use comparison operators in logical statements to find out if variables or values are different.

You get either true or false.

For example, the equal to (==) operator checks whether the operands' values are equal.

```
var num = 10;
console.log(num == 8);
```

Heads up!

You can compare all types of data with comparison operators, they'll always return true or false.

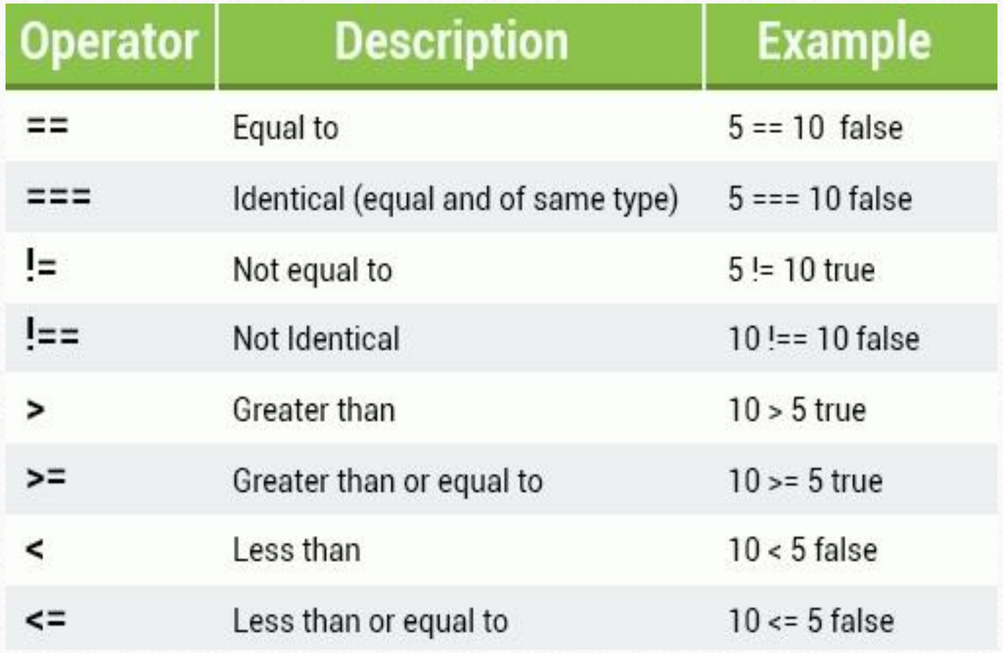
Comparison Operators

Check out this table to see a breakdown of comparison operators.

Heads up!

One important thing to remember when we use operators, is that they only work when they're comparing the same data type;

numbers with numbers, strings with strings, you get the idea.



Operator	Description	Example
==	Equal to	5 == 10 false
===	Identical (equal and of same type)	5 === 10 false
!=	Not equal to	5 != 10 true
!==	Not Identical	10 !== 10 false
>	Greater than	10 > 5 true
>=	Greater than or equal to	10 >= 5 true
<	Less than	10 < 5 false
<=	Less than or equal to	10 <= 5 false

Task

Math Operators

There are a lot of situations where you want to check someone's age...not just at the bar!

You are given a program that takes the age of the user as input.
Complete the code to check if the user is an adult, and output to the console the corresponding **boolean value**.

Sample Input
20

Sample Output
true

If the user is 18 or older, they're considered an adult.
console.log(20>18) outputs **true.**

Logical or Boolean Operators

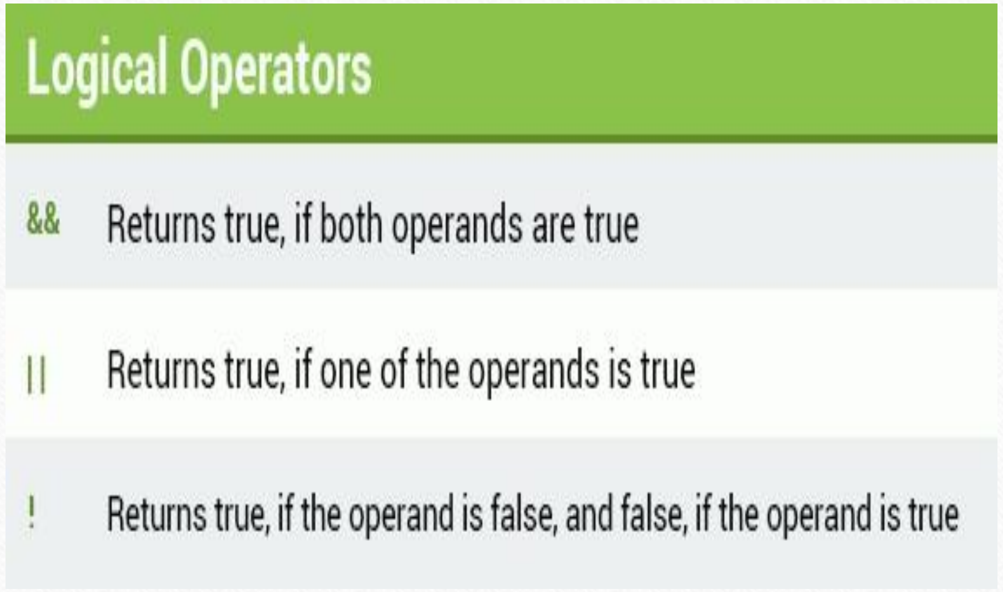
Logical Operators, also known as Boolean Operators, (or the Vulcan Operators….

ok, that one isn't true) evaluate an expression and return true or false.

Check out the table below to see more details on the logical operators (AND, OR, NOT).

Heads up!

You can check all types of data; comparison operators always return true or false.



Logical Operators

&&	Returns true, if both operands are true
||	Returns true, if one of the operands is true
!	Returns true, if the operand is false, and false, if the operand is true

Logical or Boolean Operators

Let's take a look at an example. Here we’ve connected two Boolean expressions with the AND operator.

```
document.write((4 > 2) && (10 < 15));
```

For this expression to be true, both conditions need to be true.

- The first condition determines whether 4 is greater than 2, which is true.
- The second condition determines whether 10 is less than 15, which is also true.

Ta da! The whole expression is true...very logical!

Conditional, or Ternary, operators assign a value to a variable, based on some condition.

This is what the syntax would look like:

```
variable = (condition) ? value1: value2
```

Logical or Boolean Operators

And here’s an example:

```
var age = 42;
var isAdult = (age < 18) ? "Too young": "Old enough";
document.write(isAdult);
```

If the variable age is a value below 18, the value of the variable isAdult will be "Too young".

Otherwise the value of isAdult will be "Old enough".

Heads up!

With logical operators you can connect as many expressions as you want or need to.

Task

Logical or boolean operators

Time flies when you’re having fun.
Given a clock that measures 24 hours in a day, write a program that takes the hour as input. If the hour is in the range of 0 to 12, output **am** to the console, and output **pm** if it's not.

Sample Input
13

Sample Output
pm

Assume the input number is positive and less than or equal to 24.

String Operators

Time to introduce the most useful operator for strings...drum roll please.

...Concatenation.

We can use concatenation (represented by the + sign) to build strings made up of multiple smaller strings,

or by joining strings with other types. Check it out:

```
var mystring1 = "I am learning ";
var mystring2 = "JavaScript with Google.";
document.write(mystring1 + mystring2);
```

This example declares and initializes two string variables, and then concatenates them. Simple...but super useful!

Heads up!

Numbers in quotes are treated as strings: So "42" is not the number 42, it’s a string that includes the two separate characters, 4 and 2.

The if Statement

Well done! You’re making great progress. On to module 3!

Often when we write code, we want to perform different actions based on different conditions.

And this is where conditional statements come in.

There are a bunch of different conditionals, to cover, but we’re starting with one of the most useful: "if"

We use if to specify a block of code that we want to be executed if a specified condition is true.

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

The if Statement

The statements will only be executed if the specified condition is true. Let's take a look at an example:

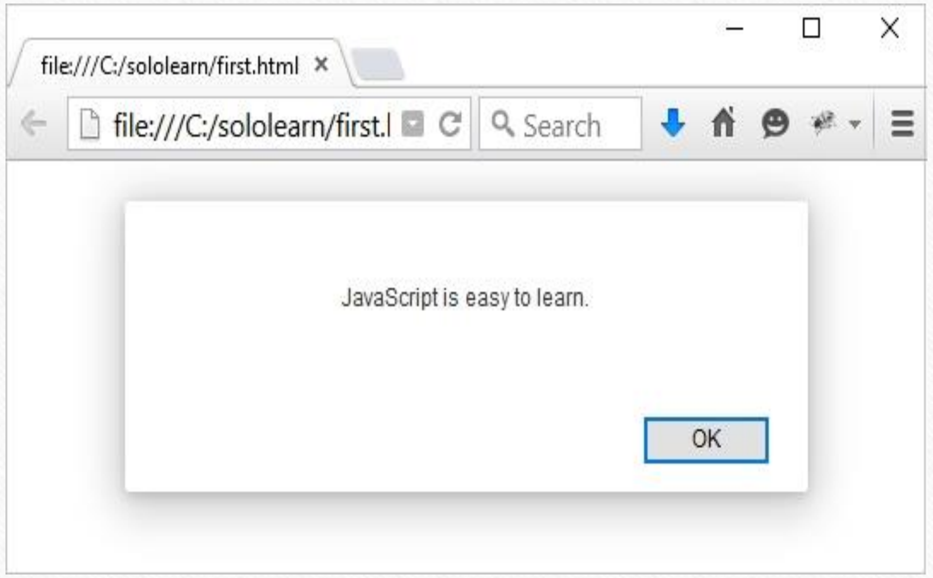
```
var myNum1 = 7;
var myNum2 = 10;
if (myNum1 < myNum2) {
   alert("JavaScript is easy to learn.");
}
```

Heads up!

You can see from the example above,

we've used the JavaScript alert() to generate a popup alert box that contains the information inside the parentheses.

Result:



JavaScript is easy to learn.

The if Statement

Here’s a little more detail on the if statement.

This is an example of a false conditional statement:

```
var myNum1 = 7;
var myNum2 = 10;
if (myNum1 > myNum2) {
   alert("JavaScript is easy to learn.");
}
```

Because the condition evaluates to false,

the alert statement gets skipped and the program continues with the line after the if statement's closing curly brace.

Heads up!

if is in lowercase letters. Uppercase letters (If or IF) won’t work.

Task

The if Statement

You are planning a vacation in August.
You are given a program that takes the month as input.

Task
Complete the code to output "**vacation**", if the given month is August. Don't output anything otherwise.

Sample Input
August

Sample Output
vacation

Handle the required condition using an **if statement**.
Use **console.log()** for the output.

The else Statement

Right, so we've seen that the action gets skipped when a code block using the if statement evaluates to false, but what if we want something else to happen. Well, we use the "else" statement, of course!

We can use the else statement to specify a block of code that will execute if the condition is false. Like this:

```
if (expression) {
   // executed if condition is true
}
else {
   // executed if condition is false
}
```

Heads up!

You can skip the curly braces if the code under the condition contains only one command.

The else Statement

Here’s another example of the if and else statements working together:

```
var myNum1 = 7;
var myNum2 = 10;
if (myNum1 > myNum2) {
   alert("This is my first condition");
}
else {
   alert("This is my second condition");
}
```

Let's translate that example. It says:

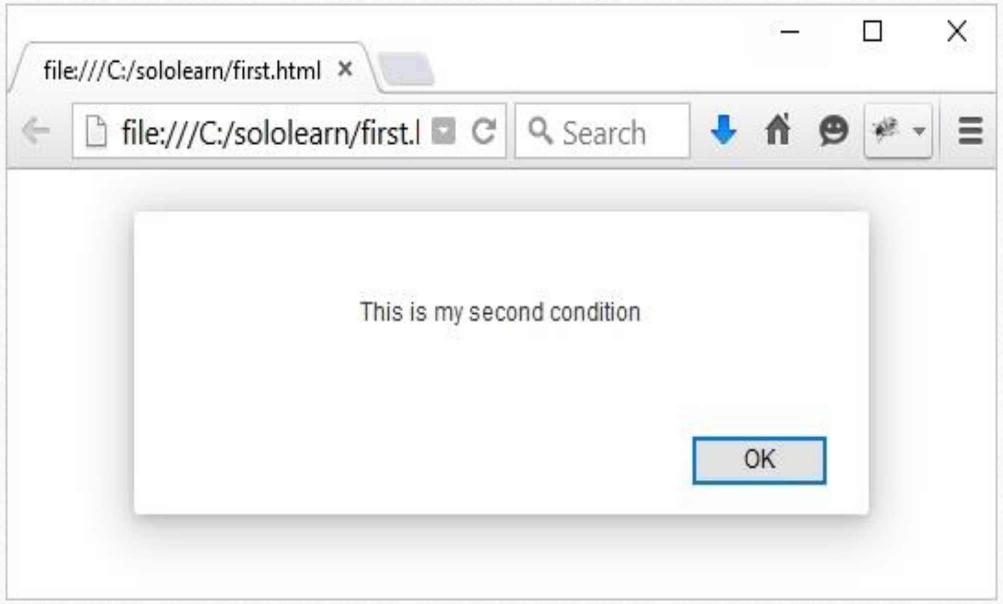
- If myNum1 is greater than myNum2, alert "This is my first condition";
- Else, alert "This is my second condition".

The else Statement

So the browser will print out the second condition, as 7 is not greater than 10.

Heads up!

There's another way to do this check using the ? operator: a > b ? alert(a) : alert(b).



This is my second condition

Task

The if else statement

The current world record for high jumping is 2.45 meters.
You are given a program that receives as input a number that represents the height of the jump.

Task
Complete the code to:
1. output to the console **"new record"** if the number is **more than 2.45**,
2. output to the console **"not this time"** in other cases.

Sample Input
2.49

Sample Output
new record

Note that if the jump height is equal to 2.45 meters, it's not a new world record.

The else if Statement

We've seen else, we've seen if, time to meet else if.

The else if statement is useful because it lets us specify a new condition if the first condition is false.

Like this:

```
var course = 1;
if (course == 1) {
   document.write("<h1>HTML Tutorial</h1>");
} else if (course == 2) {
   document.write("<h1>CSS Tutorial</h1>");
} else {
   document.write("<h1>JavaScript Tutorial</h1>");
}
```

The else if Statement

This is what's happening in the code above:

- if course is equal to 1, output "HTML Tutorial";
- else, if course is equal to 2, output "CSS Tutorial";
- if none of the above condition is true, then output "JavaScript Tutorial";

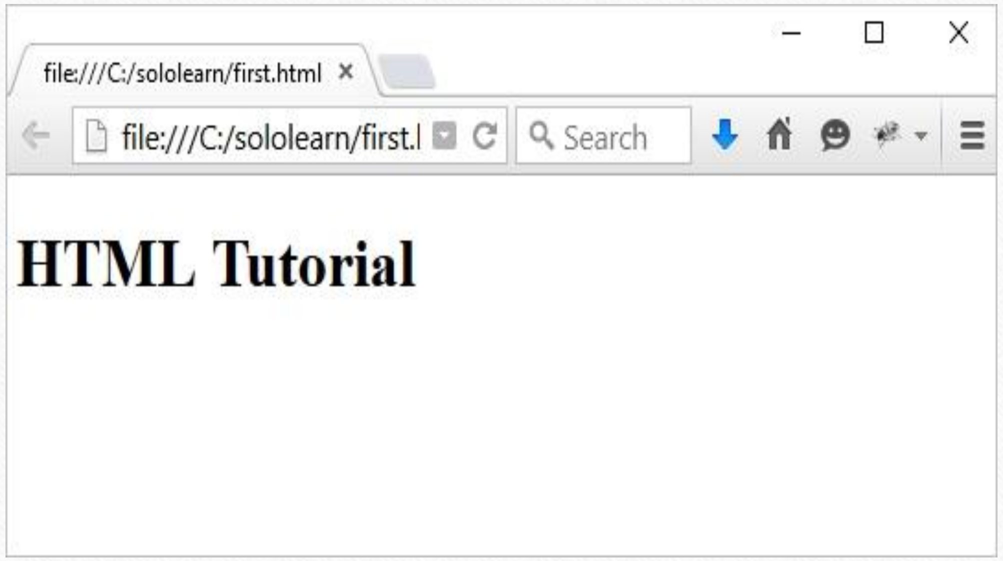
The else if Statement

course is equal to 1, so we get the following result:

Heads Up!

The final else statement "ends" the else if statement and should be always written after the if and else if statements.

The final else block will be executed when none of the conditions is true.



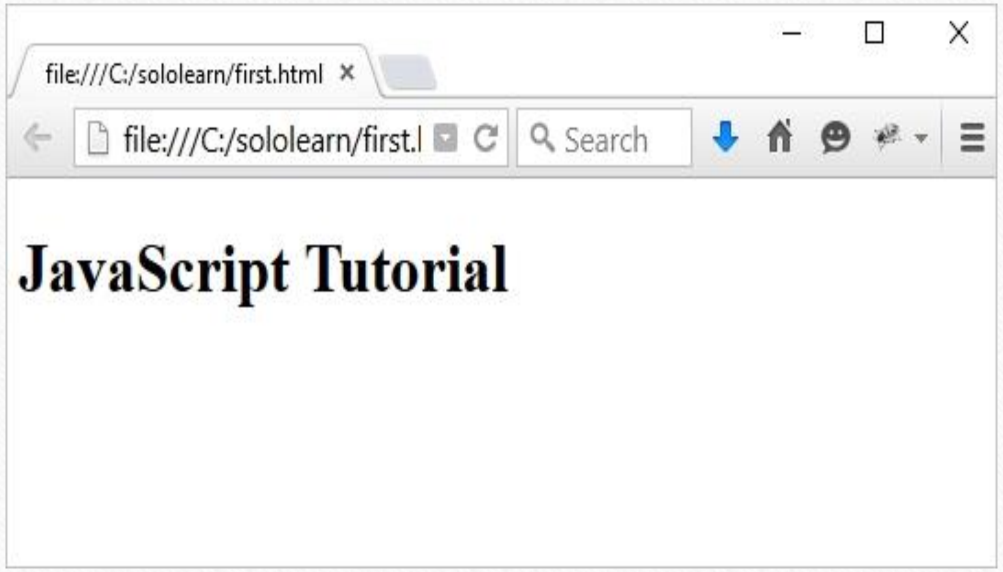
HTML Tutorial

The else if Statement

Let's change the value of the course variable in our previous example.

```
var course = 3;
if (course == 1) {
   document.write("<h1>HTML Tutorial</h1>");
} else if (course == 2) {
   document.write("<h1>CSS Tutorial</h1>");
} else {
   document.write("<h1>JavaScript Tutorial</h1>");
}
```

You can write as many else if statements as you need.



JavaScript Tutorial

Task

else if

The result of an exam will be determined as follows:
If the score is
88 and above => excellent
40-87 => good
0-39 => fail
You are given a program that takes the **score** as input.
Task
Complete the code to output the corresponding result **(excellent, good, fail)** to the console.
Sample Input
78
Sample Output
good

Use **console.log()** to output the result to the console.
Use logical operator **&&** to chain multiple conditions.

The switch Statement

But what if you need to test for multiple conditions? In those cases, writing if else statements for each condition might not be the best solution. Instead, we can use the switch statement to perform different actions based on different conditions. Here's what that looks like:

```
switch (expression) {
  case n1:
    statements
    break;
  case n2:
    statements
    break;
  default:
    statements
}
```

The switch Statement

The switch expression is evaluated once.

The value of the expression is compared with the values of each case, and if there's a match, that block of code is executed.

Heads up!

You can achieve the same result with multiple if...else statements, but the switch statement is more effective in such situations.

The switch Statement

Let's look at another example:

```
var day = 2;
switch (day) {
  case 1:
    document.write("Monday");
    break;
  case 2:
    document.write("Tuesday");
    break;
case 3:
    document.write("Wednesday");
    break;
  default:
    document.write("Another day");
}
```

Heads up!

You can have as many case statements as you need.

The switch Statement

So we have learned that the switch statement tests a code block, but we won't always want it to test the whole block.

The break keyword essentially switches the switch statement off.

Breaking out of the switch block stops the execution of more code and case testing inside the block.

Heads up!

Usually, a break should be put in each case statement.

Often there will be no match, but we still need the program to output something...for this we use the default keyword,

which specifies the code to run if there’s no case match.

The switch Statement

Like this:

```
var color ="yellow";
switch(color) {
    case "blue":
        document.write("This is blue.");
        break;
    case "red":
        document.write("This is red.");
        break;
case "green":
        document.write("This is green.");
        break;
    case "orange":
        document.write("This is orange.");
        break;
    default:
        document.write("Color not found.");
}
```

Heads up! The default block can be omitted, if there is no need to handle the case when no match is found.

Task

Switch statements

The user can choose the color theme for the browser:
1. Light
2. Dark
3. Nocturne
4. Terminal
5. Indigo

You are given a program that takes
the number as input. Complete the program so that it will output to the console the theme according to input number.

Sample Input

2

Sample Output

Dark

Don't forget to use **break** for each case statement.

The For Loop

Loops can execute a block of code a number of times.

They’re handy when you want to run the same code repeatedly, adding a different value each time.

JavaScript has three types of loops: for, while, and do while.

We’ll start here with the classic for loop.

Here's the syntax:

```
for (statement 1; statement 2; statement 3) {
  code block to be executed
}
```

The For Loop

And here’s what happens when it runs:

- Statement 1 is executed before the loop (the code block) starts.
- Statement 2 defines the condition for running the loop (the code block).
- Statement 3 is executed each time after the loop (the code block) has been executed.

Heads up!

As you can see, the classic for loop has three components, or statements.

The For Loop

Now we've got the theory, let's look at a specific example.

This example creates a for loop that prints numbers 1 through 5:.

```
for (i=1; i<=5; i++) {
    document.write(i + "<br />");
}
```

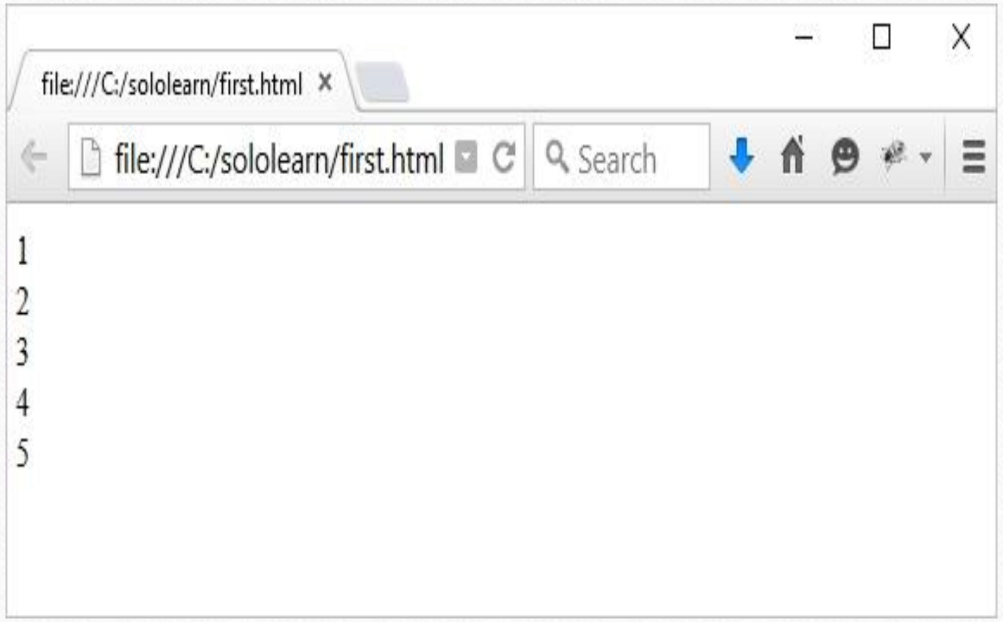
The For Loop

So what's actually happening?

In this example, Statement 1 sets a variable before the loop starts (var i = 1).

Statement 2 defines the condition for the loop to run (it must be less than or equal to 5).

Statement 3 increases a value (i++) each time the code block in the loop has been executed.



```
1
2
3
4
5
```

The For Loop

Statement 1 is optional, and can be left out, if your values are set before the loop starts.

```
var i = 1;
for (; i<=5; i++) {
   document.write(i + "<br />");
}
```

You can also initiate more than one value in statement 1, using commas to separate them.

```
for (i=1, text=""; i<=5; i++) {
   text = i;
   document.write(i + "<br />");
}
```

Heads up!

ES6 introduces other for loop types; you can learn about them in the ES6 course later.

The For Loop

If statement 2 returns true, the loop will start over again, if it returns false, the loop will end.

Statement 2 is also optional, but only if you put a break inside the loop. Otherwise, the loop will never end!

Statement 3 is used to change the initial variable. It can do anything, including negative increment (i--), positive increment (i = i + 15).

Statement 3 is also optional, but only if you increment your values inside the loop. Like this:

```
var i = 0;
for (; i < 10; ) {
    document.write(i);
    i++;
}
```

Heads up! You can have multiple nested for loops.

Task

The for loop

"Repetition is the mother of learning, the father of action, which makes it the architect of accomplishment." - Zig Ziglar.
Inspired by these words, let's create a little program that will output an expression which is given as input, 3 times.

Task
Complete the code to output the given expression 3 times.
Sample Input
Learning is fun!
Sample Output
Learning is fun!
Learning is fun!
Learning is fun!

Use the **for loop** to run the required part of the code as many times as needed.

The While Loop

Time to move on to the second of our three loop statements, while.

The while loop repeats through a block of code, but only as long as a specified condition is true.

Here's the syntax:

```
while (condition) {
   code block
}
```

Heads up!

The condition can be any conditional statement that returns true or false.

The While Loop

Ok, we've got the theory, let's look at a real example:

```
var i=0;
while (i<=10) {
    document.write(i + "<br />");
    i++;
}
```

The loop in this code will continue to run as long as i is less than, or equal to, 10. And each time the loop runs, it will increase by 1.

Heads up! Be careful when writing conditions. If a condition is always true, the loop will run forever!

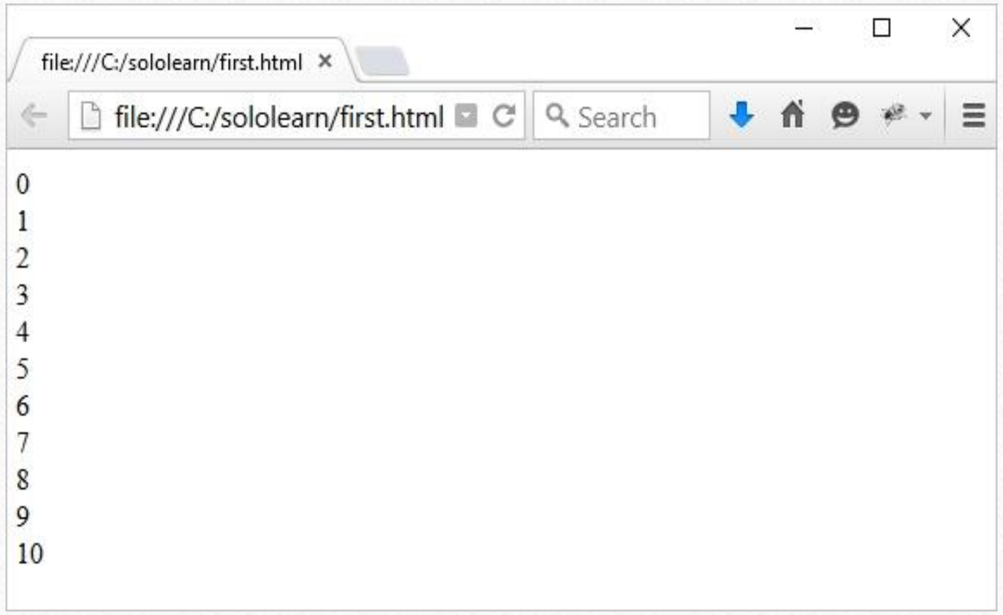
Endless loops are not good. And one way of this happening is if we forget to increase the variable used in the condition.

The While Loop

This will output the values from 0 to 10.

Heads up!

Make sure that the condition in a while loop eventually becomes false.


file:///C:/sololearn/first.html
file:///C:/sololearn/first.html
Search

```
0
1
2
3
4
5
6
7
8
9
10
```

Task

The while loop

Write a **program-timer**, that will take the count of seconds as input and output to the console all the seconds until timer stops.

Sample Input
4

Sample Output
```
4
3
2
1
0
```

Be attentive, 0 should also be included in the output.

The Do...While Loop

Almost done with loops! You're doing great!

The last loop we're looking at in this module is the do...while loop, it's a variant of the while loop but with one important difference.

This loop will execute the code block once, before checking if the condition is true,

and then it will repeat the loop as long as the condition is true.

Here's the Syntax:

```
do {
  code block
}
while (condition);
```

Heads up! Note the semicolon used at the end of the do...while loop. This is important.

The Do...While Loop

Here’s a real example:

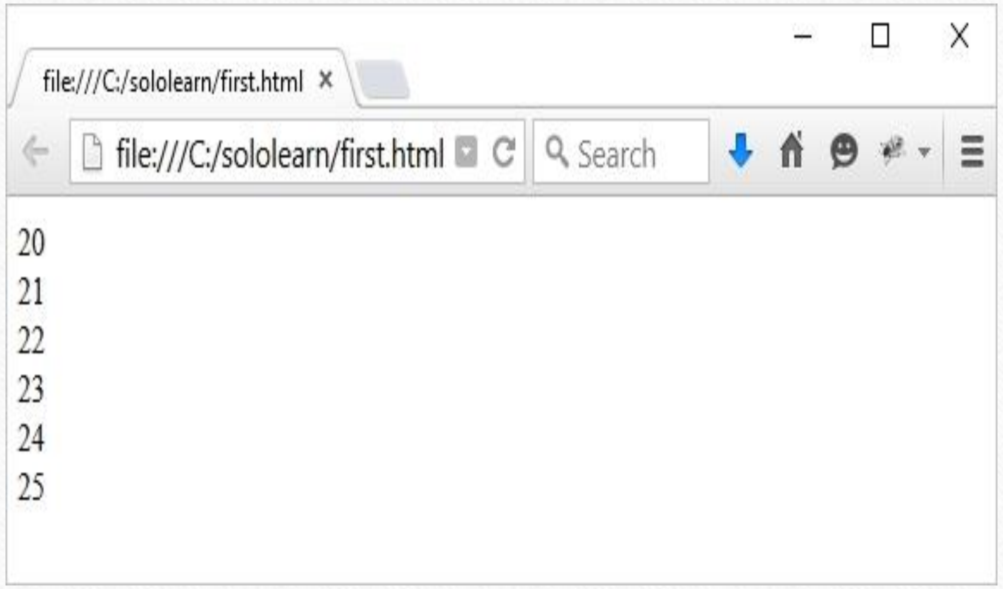
```
var i=20;
do {
    document.write(i + "<br />");
    i++;
}
while (i<=25);
```

This example prints out numbers from 20 to 25.

The Do...While Loop

Heads up!

The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested.


file:///C:/sololearn/first.html
file:///C:/sololearn/first.html
Search

```
20
21
22
23
24
25
```

Break and Continue

We've met the break statement earlier in this module, we use it to "jump out" of a loop and continue executing the code after the loop.

Like this:

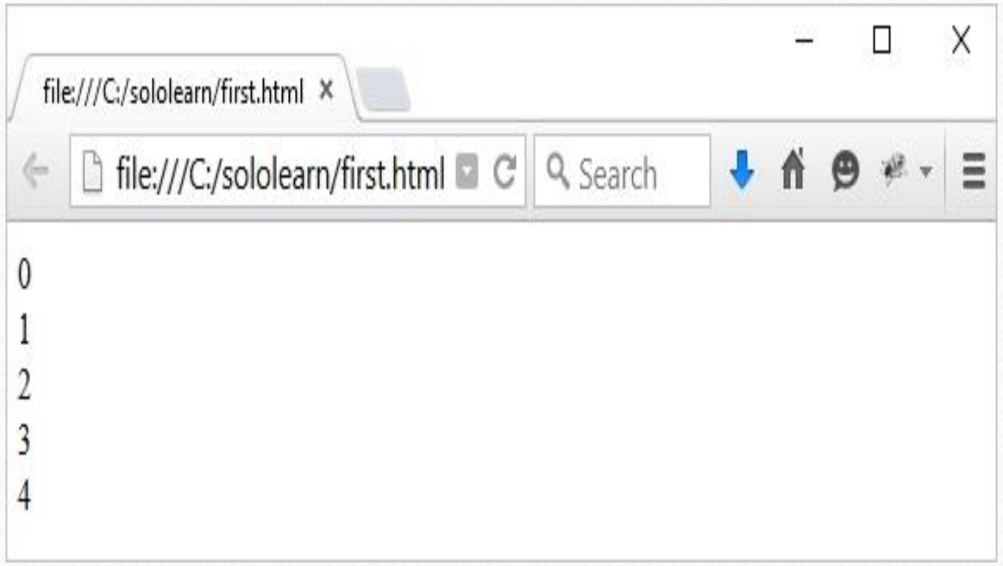
```
for (i = 0; i <= 10; i++) {
    if (i == 5) {
        break;
    }
    document.write(i + "<br />");
}
```

Break and Continue

In this example, once i reaches 5, it will break out of the loop.

Heads up!

You can use the return keyword to return some value immediately from the loop inside of a function. This will also break the loop.


file:///C:/sololearn/first.html
file:///C:/sololearn/first.html
Search

```
0
1
2
3
4
```

Break and Continue

We're nearly done with module 3! One last thing to cover.

Unlike the break statement, the continue statement breaks only one iteration in the loop, and continues with the next iteration.

Like this:

```
for (i = 0; i <= 10; i++) {
    if (i == 5) {
        continue;
    }
    document.write(i + "<br />");
}
```

Heads up! The value 5 is not printed, because continue skips that iteration of the loop.

Task

Break and continue

Many tall buildings, including hotels, skip the number 13 when numbering floors -- often going from floor 12 to floor 14. It is thought that the number 13 is unlucky. Write a program that will number 15 rooms starting from 1, skipping the number 13. Output to the console each room number in separate line.

Be attentive - considering the missing number, the number of last room should be greater than the count of rooms by 1.

User-Defined Functions

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

The main advantages of using functions:

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

Use the same code many times with different arguments, to produce different results.

A JavaScript function is executed when "something" invokes, or calls, it.

To define a JavaScript function, use the function keyword, followed by a name, followed by a set of parentheses ().

The code to be executed by the function is placed inside curly brackets {}.

```
function name() {
  //code to be executed
}
```

User-Defined Functions

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

To execute the function, you need to call it.

To call a function, start with the name of the function, then follow it with the arguments in parentheses.

Example:

```
function myFunction() {
    alert("Calling a Function!");
}
myFunction();
```

Always remember to end the statement with a semicolon after calling the function.

User-Defined Functions

Once the function is defined, JavaScript allows you to call it as many times as you want to.

```
function myFunction() {
   alert("Alert box!");
}
myFunction();
// some other code
myFunction();
```

You can also call a function using this syntax: myFunction.call().

The difference is that when calling in this way, you're passing the 'this' keyword to a function. You'll learn about it later.

Task

User-Defined Functions

It’s very common to have "Preloader" component, especially in multifunctional apps and websites.
Create a function that will output "Loading" to the console.

Output
Loading

Don't forget to call the function.

Function Parameters

Functions can take parameters.

Function parameters are the names listed in the function's definition.

Syntax:

```
functionName(param1, param2, param3) {
  // some code
}
```

As with variables, parameters should be given names, which are separated by commas within the parentheses.

Function Parameters

After defining the parameters, you can use them inside the function.

```
function sayHello(name) {
    alert("Hi, " + name);
}
sayHello("David");
```

This function takes in one parameter, which is called name.

When calling the function, provide the parameter's value (argument) inside the parentheses.

Function arguments are the real values passed to (and received by) the function.

Function Parameters

You can define a single function, and pass different parameter values (arguments) to it.

```
function sayHello(name) {
    alert("Hi, " + name);
}
sayHello("David");
sayHello("Sarah");
sayHello("John");
```

This will execute the function's code each time for the provided argument.

Task

Function Parameters

Sometimes it’s very useful to set reminder to help you accomplish all of your tasks.
The program you are given takes an event as input.
Complete the function-reminder to take that event as argument and output the corresponding message.

Sample Input
workout

Sample Output
You set a reminder about workout

Use **+ operator** to concatenate string values together.
For example, console.log("JS is " + "fun!") will output "JS is fun".

Using Multiple Parameters with Functions

You can define multiple parameters for a function by comma-separating them.

```
function myFunc(x, y) {
  // some code
}
```

The example above defines the function myFunc to take two parameters.

The parameters are used within the function's definition.

```
function sayHello(name, age) {
 document.write( name + " is " + age + " years old.");
}
```

Function parameters are the names listed in the function definition.

Using Multiple Parameters with Functions

When calling the function, provide the arguments in the same order in which you defined them.

```
function sayHello(name, age) {
    document.write( name + " is " + age + " years old.");
}
sayHello("John", 20)
```

Using Multiple Parameters with Functions

If you pass more arguments than are defined, they will be assigned to an array called arguments.

They can be used like this: arguments[0], arguments[1], etc.

After defining the function, you can call it as many times as needed.

JavaScript functions do not check the number of arguments received.

If a function is called with missing arguments (fewer than declared), the missing values are set to undefined,

which indicates that a variable has not been assigned a value.

Task

Multiple Parameters

You are given a program that takes Team 1 and Team 2 football teams goals as inputs accordingly.
Complete the function to take Team 1 and Team 2 goals **as arguments** and output the final result of the match:
- **"Team 1 won"**, if Team 1's score is higher than Team 2's score
- **"Team 2 won"**, if Team 2's score is higher than Team 1's score
- **"Draw"**, if the scores are equal

Sample Input
```
3
4
```

Sample Output
```
Team 2 won
```

The function call is already given.

The return Statement

A function can have an optional return statement. It is used to return a value from the function.

This statement is useful when making calculations that require a result. When JavaScript reaches a return statement, the function stops executing. Use the return statement to return a value.

For example, let's calculate the product of two numbers, and return the result.

```
function product(a, b) {
    return a * b;
}
var result= product(8, 11);
console.log(result);
```

If you do not return anything from a function, it will return undefined.

The return Statement

Another example:

```
function addNumbers(a, b) {
    var c = a+b;
    return c;
}
document.write( addNumbers(40, 2) );
```

The document.write command outputs the value returned by the function, which is the sum of the two parameters.

Task

The return Statement

You are given a program that takes 3 numbers as input.
Complete the given function to calculate the average of those 3 numbers, assign it to the given variable, and output it.

Sample Input
3
6
9

Sample Output
6

Use return statement to **return** the calculated value and use it through the program.

Alert, Prompt, Confirm

JavaScript offers three types of popup boxes, the Alert, Prompt, and Confirm boxes.

Alert Box

An alert box is used when you want to ensure that information gets through to the user.

When an alert box pops up, the user must click OK to proceed.

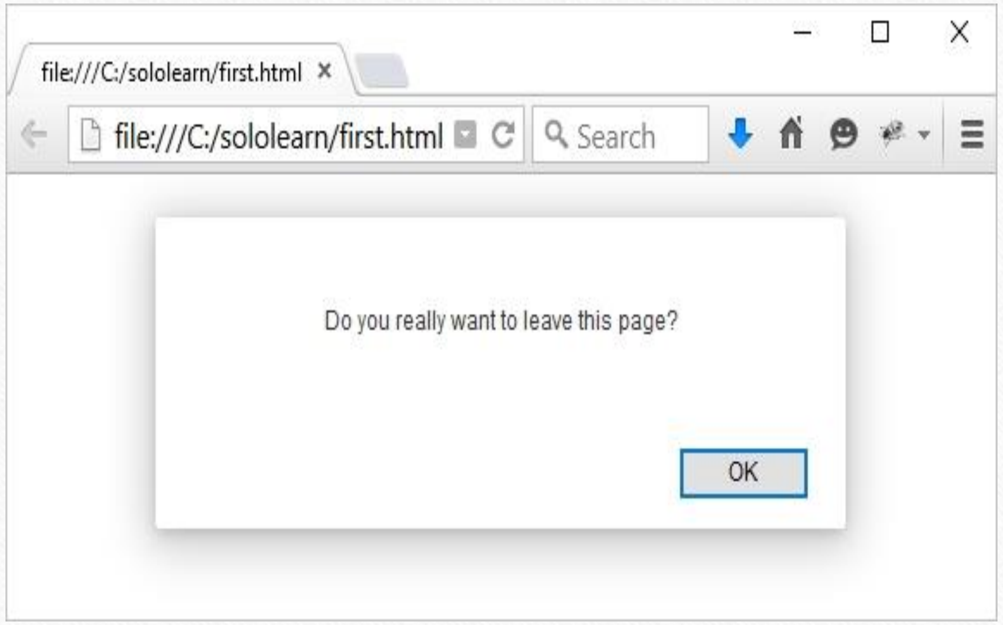
The alert function takes a single parameter, which is the text displayed in the popup box.

Alert, Prompt, Confirm

Example:

alert("Do you really want to leave this page?");

Result:



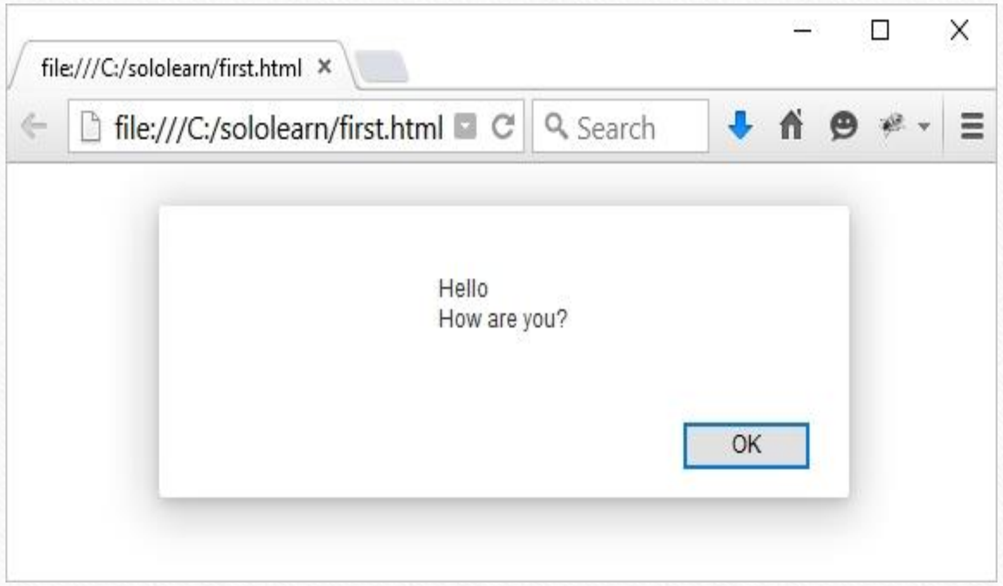
Do you really want to leave this page?

Alert, Prompt, Confirm

To display line breaks within a popup box, use a backslash followed by the character n.

alert("Hello\nHow are you?");

Result:


file:///C:/sololearn/first.html
file:///C:/sololearn/first.html
Search
OK

```
Hello
How are you?
```

Alert, Prompt, Confirm

Be careful when using alert boxes, as the user can continue using the page only after clicking OK.

A prompt box is often used to have the user input a value before entering a page.

When a prompt box pops up, the user will have to click either OK or Cancel to proceed after entering the input value.

If the user clicks OK, the box returns the input value. If the user clicks Cancel, the box returns null.

The prompt() method takes two parameters.

- The first is the label, which you want to display in the text box.
- The second is a default string to display in the text box (optional).

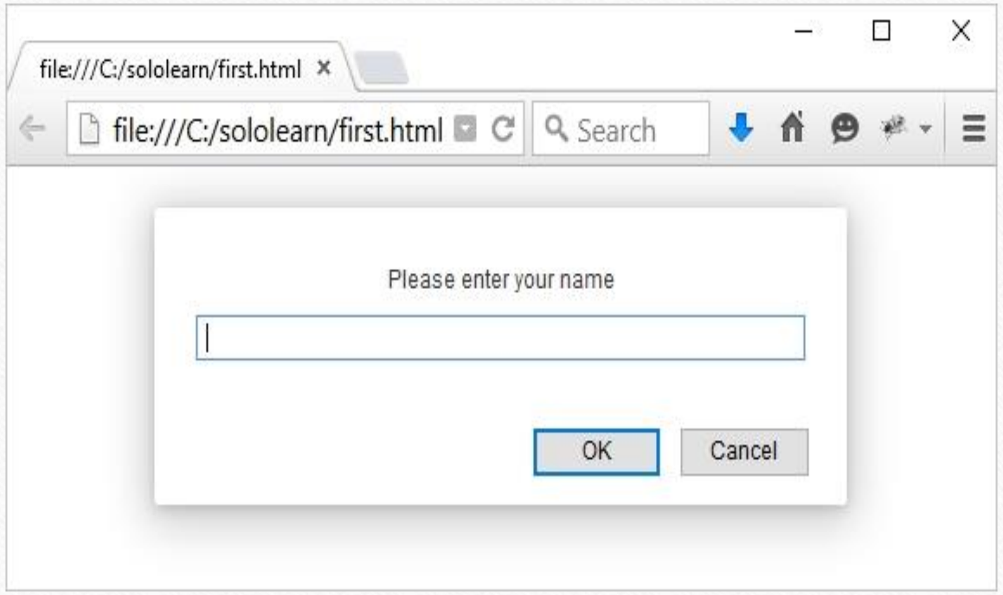
Alert, Prompt, Confirm

Example:

var user = prompt("Please enter your name");

alert(user);

The prompt appears as:



Please enter your name

Alert, Prompt, Confirm

When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value.

Do not overuse this method, because it prevents the user from accessing other parts of the page until the box is closed.

A confirm box is often used to have the user verify or accept something.

When a confirm box pops up, the user must click either OK or Cancel to proceed.

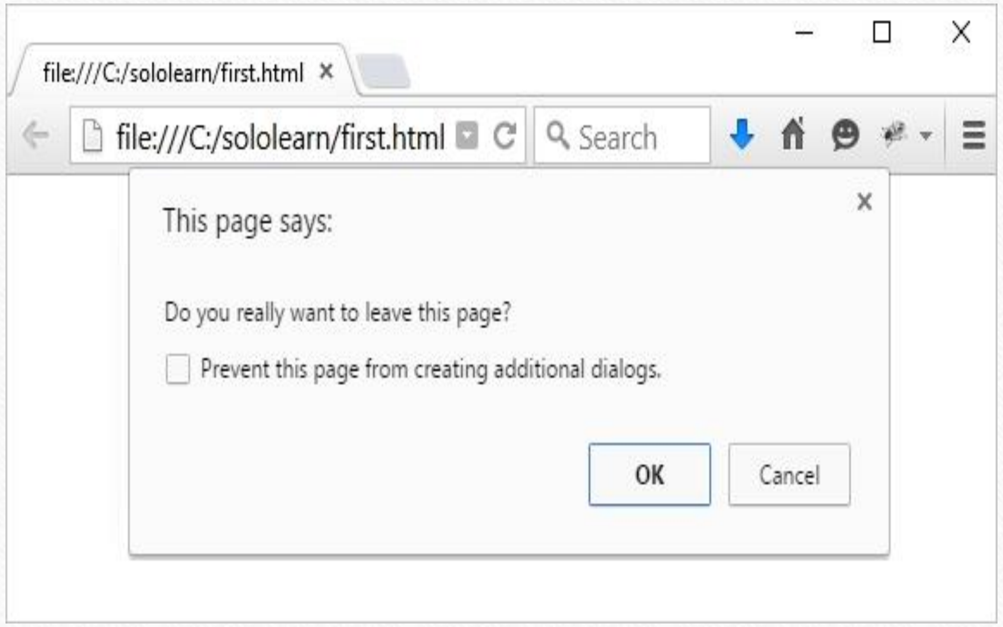
If the user clicks OK, the box returns true. If the user clicks Cancel, the box returns false.

Alert, Prompt, Confirm

Example:

```
var result = confirm("Do you really want to leave this page?");
if (result == true) {
   alert("Thanks for visiting");
}
else {
   alert("Thanks for staying with us");
}
```

Result:



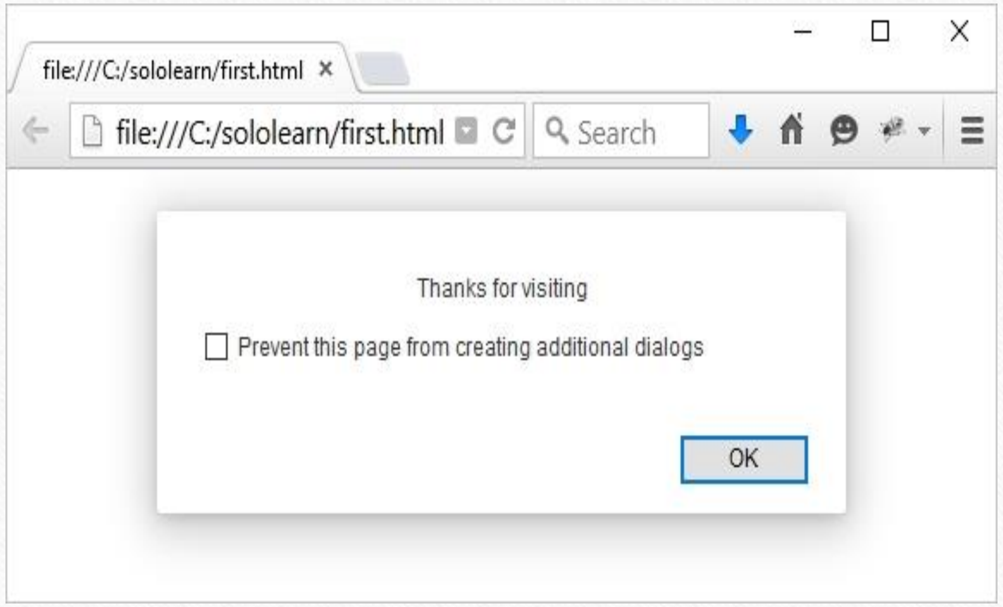
This page says:

Do you really want to leave this page?

☐ Prevent this page from creating additional dialogs.

Alert, Prompt, Confirm

The result when the user clicks OK:


file:///C:/sololearn/first.html

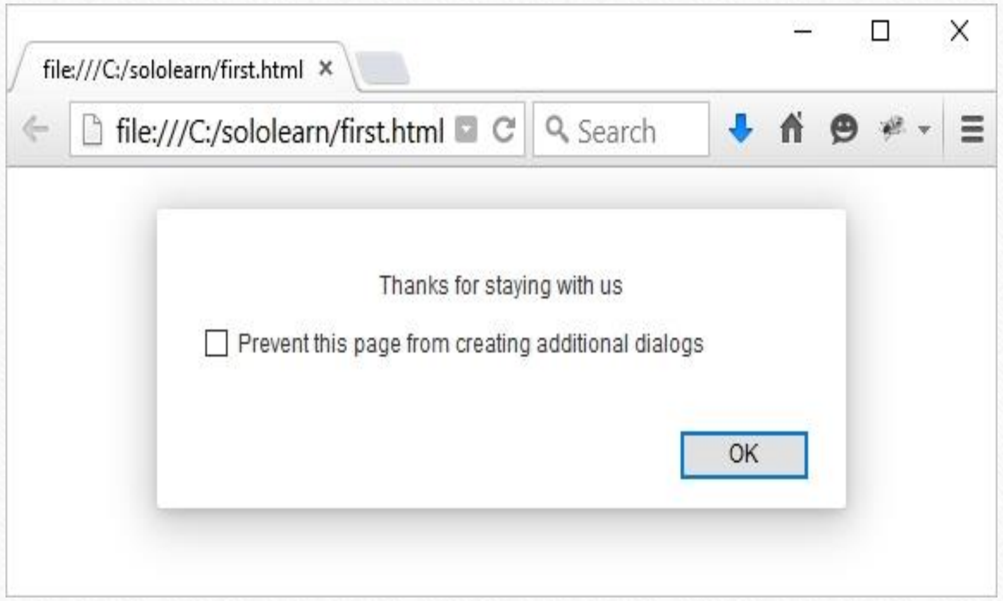
Thanks for visiting

Prevent this page from creating additional dialogs

Alert, Prompt, Confirm

The result when the user clicks Cancel:

Do not overuse this method, because it also prevents the user from accessing other parts of the page until the box is closed.


file:///C:/sololearn/first.html
file:///C:/sololearn/first.html
Search

Thanks for staying with us

☐ Prevent this page from creating additional dialogs

Introducing Objects

JavaScript variables are containers for data values. Objects are variables too, but they can contain many values.

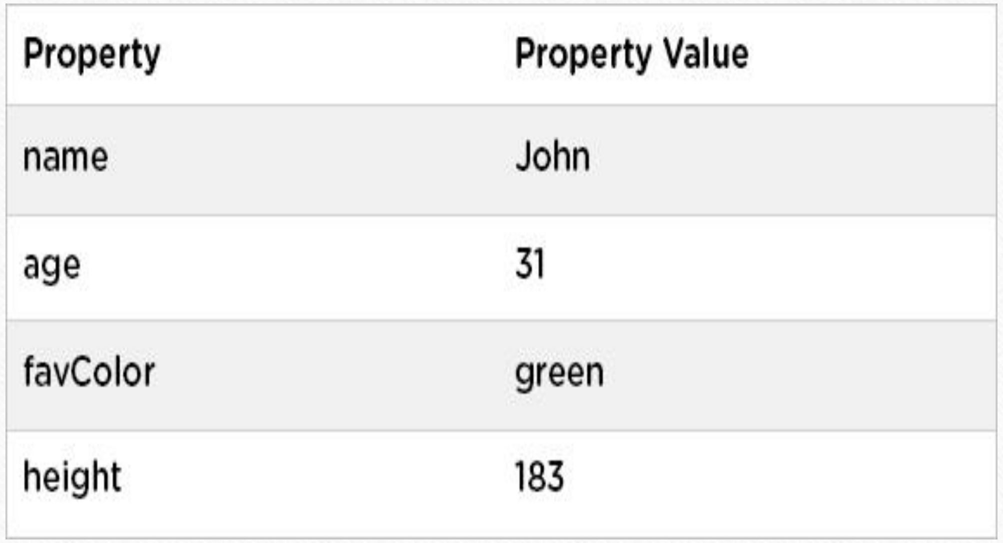
Think of an object as a list of values that are written as name:value pairs, with the names and the values separated by colons.

Example:

```
var person = {
 name: "John", age: 31,
 favColor: "green", height: 183
};
```

Introducing Objects

These values are called properties.



Property	Property Value
name	John
age	31
favColor	green
height	183

Introducing Objects

JavaScript objects are containers for named values.

You can access object properties in two ways.

```
objectName.propertyName
//or
objectName['propertyName']
```

Introducing Objects

This example demonstrates how to access the age of our person object.

```
var person = {
    name: "John", age: 31,
    favColor: "green", height: 183
};
var x = person.age;
var y = person['age'];
document.write(x);
document.write(y);
```

Introducing Objects

JavaScript's built-in length property is used to count the number of characters in a property or string.

```
var course = {name: "JS", lessons: 41};
document.write(course.name.length);
```

Objects are one of the core concepts in JavaScript.

An object method is a property that contains a function definition.

Use the following syntax to access an object method.

```
objectName.methodName()
```

As you already know, document.write() outputs data. The write() function is actually a method of the document object.

```
document.write("This is some text");
```

Methods are functions that are stored as object properties.

Task

Introducing Objects

The given class represents a cuboid (e.g. a rectangular prism) that contains the properties of length, width, and height.
Complete the program to calculate and output the volume of given cuboid to the console.

To calculate the volume of cuboid use **length*width*height** formula.

Creating Your Own Objects

In the previous lesson, we created an object using the **object literal** (or initializer) syntax.

```
var person = {
name: "John", age: 42, favColor: "green"
};
```

This allows you to create only a single object.
Sometimes, we need to set an "**object type**" that can be used to create a number of objects of a single type.
The standard way to create an "object type" is to use an object **constructor function**.

```
function person(name, age, color) {
this.name = name;
this.age = age;
this.favColor = color;
}
```

The above function (person) is an object constructor, which takes parameters and assigns them to the object properties.

The **this** keyword refers to the **current object**.
Note that **this** is not a variable. It is a keyword, and its value cannot be changed.

Creating Your Own Objects

Once you have an object constructor, you can use the **new** keyword to create new objects of the same type.

```
function person(name, age, color) {
    this.name = name;
    this.age = age;
    this.favColor = color;
}
var p1 = new person("John", 42, "green");
var p2 = new person("Amy", 21, "red");
document.write(p1.age);
document.write(p2.name);
```

p1 and *p2* are now objects of the **person** type. Their properties are assigned to the corresponding values.

```
this.age = age;
}
var John = new person("John", 25);
var James = new person("James", 21);
```

Creating Your Own Objects

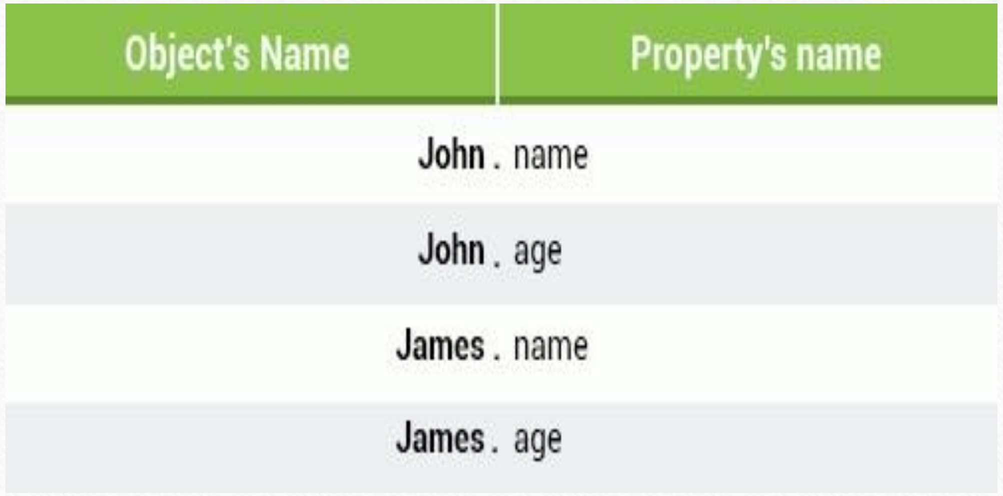
Consider the following example.

```
function person (name, age) {
    this.name = name;
    this.age = age;
}
var John = new person("John", 25);
var James = new person("James", 21);
document.write(John.age);
```

Creating Your Own Objects

Access the object's properties by using the **dot syntax**, as you did before.

Understanding the creation of objects is essential.



Object's Name	Property's name
John .	name
John .	age
James .	name
James .	age

Task

Creating Your Own Objects

Bob was hired as an airport information officer and needs to generate status messages for each flight. Let's help him!
Complete the given program by fixing the constructor, making a flight object, and assign it to given variable to correctly execute the corresponding message.
Flight ID and the flight status(landed, on time, delayed, etc.) are taken as input.

Sample Input

```
NGT 929
landed
```

Sample Output

```
Flight NGT 929 has landed
```

Use **new** keyword to create a new object using constructor.

```
this.age = age;
}
var John = new person("John", 25);
var James = new person("James", 21);
```

Object Initialization

Use the **object literal** or **initializer** syntax to create single objects.

```
var John = {name: "John", age: 25};
var James = {name: "James", age: 21};
```

Objects consist of properties, which are used to describe an object. Values of object properties can either contain primitive data types or other objects.

Object Initialization

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

```
var John = {
name: "John",
age: 25
};
var James = {
name: "James",
age: 21
};
```

No matter how the object is created, the syntax for accessing the properties and methods does not change.

Object Initialization

```
var John = {
    name: "John",
    age: 25
};
var James = {
    name: "James",
    age: 21
};
document.write(John.age);
```

Don't forget about the second accessing syntax: John['age'].

Adding Methods

Methods are functions that are stored as object properties.
Use the following syntax to create an object method:

methodName = function() { code lines }

Access an object method using the following syntax:

objectName.methodName()

A method is a function, belonging to an object. It can be referenced using the **this** keyword.
The **this** keyword is used as a reference to the current object, meaning that you can access the objects properties and methods using it.

Defining methods is done inside the constructor function.

Adding Methods

```
function person(name, age) {
    this.name = name;
    this.age = age;
    this.changeName = function (name) {
        this.name = name;
    }
}
var p = new person("David", 21);
p.changeName("John");
document.write(p.name);
```

Adding Methods

In the example above, we have defined a method named **changeName** for our person, which is a function, that takes a parameter **name** and assigns it to the **name** property of the object.
this.name refers to the name property of the object.

The **changeName** method changes the object's **name** property to its argument.

Adding Methods

You can also define the function outside of the constructor function and associate it with the object.

```
function person(name, age) {
this.name= name;
this.age = age;
this.yearOfBirth = bornYear;
}
function bornYear() {
return 2016 - this.age;
}
```

As you can see, we have assigned the object's **yearOfBirth** property to the **bornYear** function.
The **this** keyword is used to access the *age* property of the object, which is going to call the method.

Note that it's not necessary to write the function's parentheses when assigning it to an object.

Adding Methods

Call the method as usual.

```
function person(name, age) {
   this.name= name;
   this.age = age;
   this.yearOfBirth = bornYear;
}
function bornYear() {
   return 2016 - this.age;
}
var p = new person("A", 22);
document.write(p.yearOfBirth());
```

Call the method by the **property name** you specified in the constructor function, rather than the function name.

Task

Adding Methods

A store manager needs a program to set discounts for products.
The program should take the product ID, price and discount as input and output the discounted price. However, the **changePrice** method, which should calculate the discount, is incomplete. Fix it!

Sample Input

```
LD1493
1700
15
```

Sample Output

```
LD1493 price: 1700
LD1493 new price: 1445
```

The first input is the product ID, the second is the price before discounting, and the third is discount percentage.
So after discounting the new price will be 1700-(0.15*1700) = **1445**.

Arrays

Arrays store multiple values in a single variable.

To store three course names, you need three variables.

```
var course1 ="HTML";
var course2 ="CSS";
var course3 ="JS";
```

But what if you had 500 courses? The solution is an array

```
var courses = new Array("HTML", "CSS", "JS");
```

This syntax declares an array named courses, which stores three values, or elements.

Arrays

You refer to an array element by referring to the index number written in square brackets.

This statement accesses the value of the first element in courses and changes the value of the second element.

```
var courses = new Array("HTML", "CSS", "JS");
var course = courses[0]; // HTML
courses[1] = "C++"; //Changes the second element
```

[0] is the first element in an array. [1] is the second. Array indexes start with 0.

Attempting to access an index outside of the array, returns the value undefined

```
var courses = new Array("HTML", "CSS", "JS");
document.write(courses[10]);
```

Our courses array has just 3 elements, so the 10th index, which is the 11th element, does not exist (is undefined).

Task

Arrays

The array you are given represents the menu of breakfast options available at the hotel.
The Chef decided to replace one of the options with "Fluffy Pancakes".
Write a program to take the index as input, replace the element with that index with "Fluffy Pancakes", and output the new menu to the console as an array.

Sample Input

```
2
```

Sample Output

```
[
'Cinnamon Doughnuts',
'Waffles',
'Fluffy Pancakes',
'Chorizo Burrito',
'French Toast'
]
```

The element with index 2 has been replaced in the output array.

Remember that the **first element** of an array has **0 index**.

Other Ways to Create Arrays

You can also declare an array, tell it the number of elements it will store, and add the elements later.

```
var courses = new Array(3);
courses[0] = "HTML";
courses[1] = "CSS";
courses[2] = "JS";
document.write(courses[2]);
```

An array is a special type of object.

An array uses numbers to access its elements, and an object uses names to access its members.

Other Ways to Create Arrays

JavaScript arrays are dynamic, so you can declare an array and not pass any arguments with the Array() constructor.

You can then add the elements dynamically.

```
var courses = new Array();
courses[0] = "HTML";
courses[1] = "CSS";
courses[2] = "JS";
courses[3] = "C++";
document.write(courses[2]);
```

You can add as many elements as you need to.

Other Ways to Create Arrays

Array Literal

For greater simplicity, readability, and execution speed, you can also declare arrays using the array literal syntax.

```
var courses = ["HTML", "CSS", "JS"];
document.write(courses[2]);
```

This results in the same array as the one created with the new Array() syntax.

You can access and modify the elements of the array using the index number, as you did before.

The array literal syntax is the recommended way to declare arrays.

Array Properties & Methods

JavaScript arrays have useful built-in properties and methods.

An array's length property returns the number of it's elements.

```
var courses = ["HTML", "CSS", "JS"];
document.write(courses.length);
```

The length property is always one more than the highest array index.

If the array is empty, the length property returns 0.

Array Properties & Methods

JavaScript's concat() method allows you to join arrays and create an entirely new array.

Example:

```
var c1 = ["HTML", "CSS"];
var c2 = ["JS", "C++"];
var courses = c1.concat(c2);
document.write(courses[2]);
```

The courses array that results contains 4 elements (HTML, CSS, JS, C++).

The concat operation does not affect the c1 and c2 arrays - it returns the resulting concatenation as a new array.

Task

Array Properties & Methods

The player receives points after passing each level of a game.
The program given takes the number of passed levels as input, followed by the points gained for each level, and creates the corresponding array of points.
Complete the program to calculate and output to the console the sum of all gained points.
Sample Input
3
1
4
8

Sample Output
13
Explanation
The first input represents the number of passed levels, -- in this case, 3 (the size of an array to be created). The next 3 inputs are the points awarded to the player for passing each level. The player gained 1+4+8 points for 3 passed levels, which is then output.
Notice that the first inputted number can be used as length of an array.

Associative Arrays

While many programming languages support arrays with named indexes (text instead of numbers), called associative arrays JavaScript does not.

However, you still can use the named array syntax, which will produce an object.

For example:

```
var person = []; //empty array
person["name"] = "John";
person["age"] = 46;
document.write(person["age"]);
```

Now, person is treated as an object, instead of being an array.

The named indexes "name" and "age" become properties of the person object.

Associative Arrays

As the person array is treated as an object, the standard array methods and properties will produce incorrect results.

For example, person.length will return 0.

Remember that JavaScript does not support arrays with named indexes.

In JavaScript, arrays always use numbered indexes.

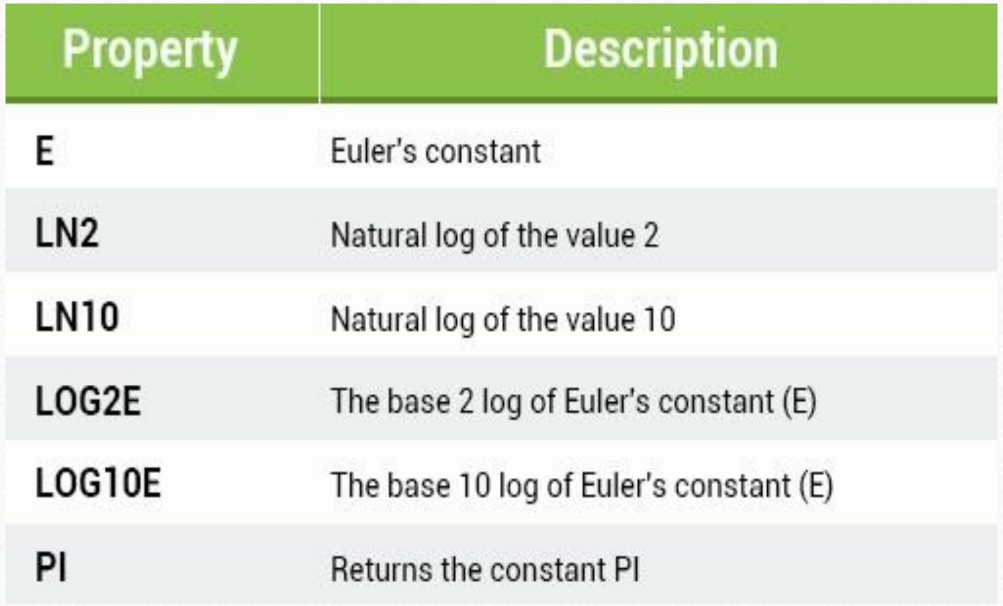
It is better to use an object when you want the index to be a string (text).

Use an array when you want the index to be a number.

If you use a named index, JavaScript will redefine the array to a standard object.

The Math Object

The Math object allows you to perform mathematical tasks, and includes several properties.



Property	Description
E	Euler's constant
LN2	Natural log of the value 2
LN10	Natural log of the value 10
LOG2E	The base 2 log of Euler's constant (E)
LOG10E	The base 10 log of Euler's constant (E)
PI	Returns the constant PI

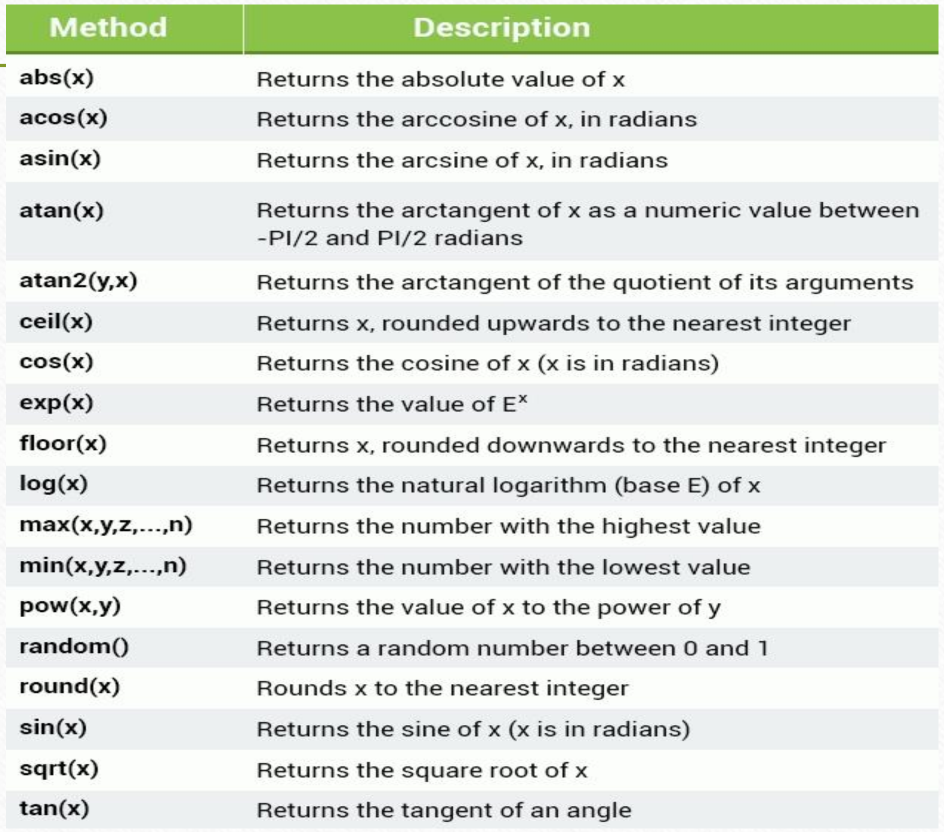
The Math Object

For example:

document.write(Math.PI);

Math has no constructor. There's no need to create a Math object first.

The Math object contains a number of methods that are used for calculations:



Method	Description
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x, in radians
asin(x)	Returns the arcsine of x, in radians
atan(x)	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians
atan2(y,x)	Returns the arctangent of the quotient of its arguments
ceil(x)	Returns x, rounded upwards to the nearest integer
cos(x)	Returns the cosine of x (x is in radians)
exp(x)	Returns the value of E^x
floor(x)	Returns x, rounded downwards to the nearest integer
log(x)	Returns the natural logarithm (base E) of x
max(x,y,z,...,n)	Returns the number with the highest value
min(x,y,z,...,n)	Returns the number with the lowest value
pow(x,y)	Returns the value of x to the power of y
random()	Returns a random number between 0 and 1
round(x)	Rounds x to the nearest integer
sin(x)	Returns the sine of x (x is in radians)
sqrt(x)	Returns the square root of x
tan(x)	Returns the tangent of an angle

The Math Object

For example, the following will calculate the square root of a number.

```
var number = Math.sqrt(4);

document.write(number);
```

To get a random number between 1-10, use Math.random(), which gives you a number between 0-1.

Then multiply the number by 10, and then take Math.ceil() from it: Math.ceil(Math.random() * 10).

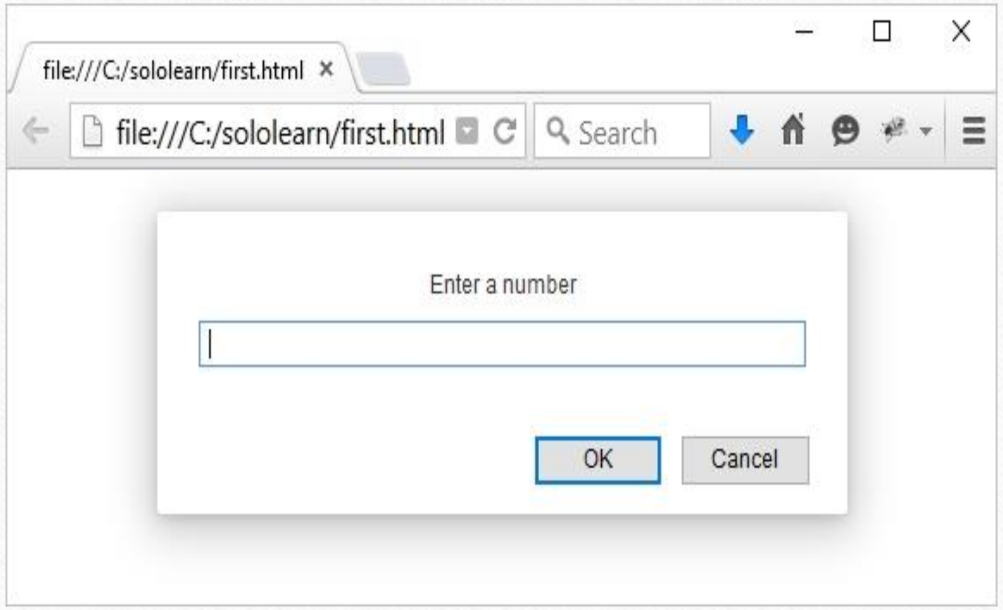
Let's create a program that will ask the user to input a number and alert its square root.

```
var n = prompt("Enter a number", "");

var answer = Math.sqrt(n);

alert("The square root of " + n + " is " + answer);
```

Math is a handy object. You can save a lot of time using Math, instead of writing your own functions every time.


file:///C:/sololearn/first.html
file:///C:/sololearn/first.html
Search
Enter a number
OK
Cancel

Task

The Math Object

Create a function that returns the century depending on the year given as a parameter.

Sample Input
1993

Sample Output
20

Hint
You need to divide 1993 by 100: 1993/100 = 19.93, then round it to the nearest integer, which is 20 in this case.

Use **Math.ceil(x)** method, which returns x rounded upwards to the nearest integer.

The Date Object

The setInterval() method calls a function or evaluates an expression at specified intervals (in milliseconds).

It will continue calling the function until clearInterval() is called or the window is closed.

For example:

```
function myAlert() {
    alert("Hi");
}
setInterval(myAlert, 3000);
```

This will call the myAlert function every 3 seconds (1000 ms = 1 second).

The Date Object

Write the name of the function without parentheses when passing it into the setInterval method.

The Date object enables us to work with dates.

A date consists of a year, a month, a day, an hour, a minute, a second, and milliseconds.

Using new Date(), create a new date object with the current date and time

```
var d = new Date();

//d stores the current date and time
```

The Date Object

The other ways to initialize dates allow for the creation of new date objects from the specified date and time

new Date(milliseconds)

new Date(dateString)

new Date(year, month, day, hours, minutes, seconds, milliseconds)

JavaScript dates are calculated in milliseconds from 01 January, 1970 00:00:00 Universal Time (UTC). One day contains 86,400,000 millisecond.

The Date Object

For example:

//Fri Jan 02 1970 00:00:00

var d1 = new Date(86400000);

//Fri Jan 02 2015 10:42:00

var d2 = new Date("January 2, 2015 10:42:00");

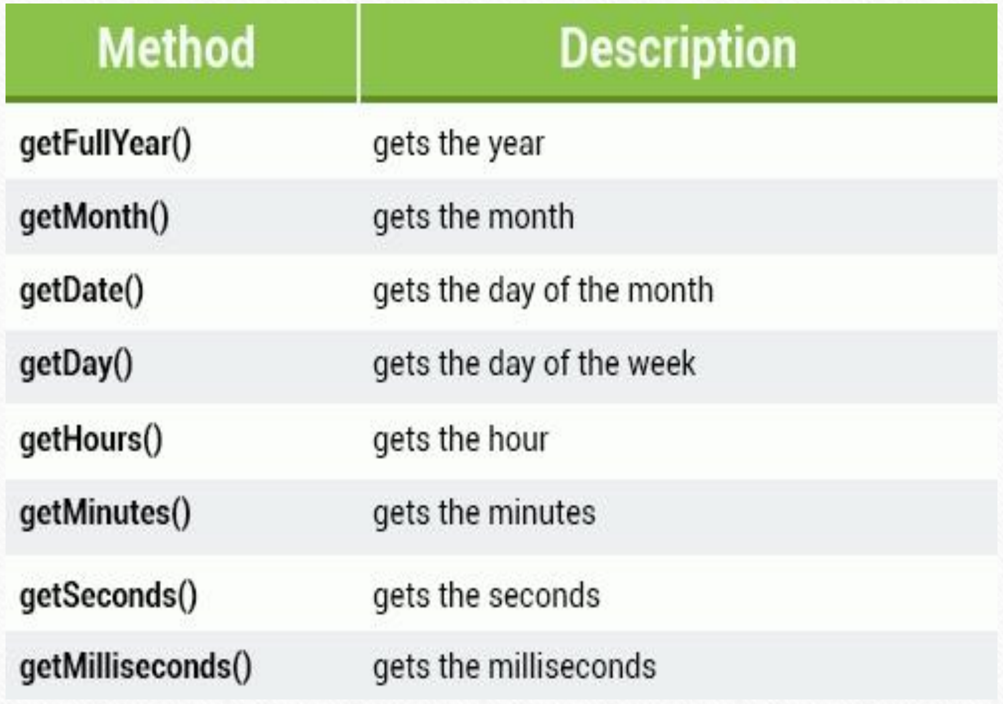
//Sat Jun 11 1988 11:42:00

var d3 = new Date(88,5,11,11,42,0,0);

JavaScript counts months from 0 to 11. January is 0, and December is 11.

Date objects are static, rather than dynamic. The computer time is ticking, but date objects don't change, once created.

When a Date object is created, a number of methods make it possible to perform operations on it.



Method	Description
getFullYear()	gets the year
getMonth()	gets the month
getDate()	gets the day of the month
getDay()	gets the day of the week
getHours()	gets the hour
getMinutes()	gets the minutes
getSeconds()	gets the seconds
getMilliseconds()	gets the milliseconds

The Date Object

For example:

```
var d = new Date();
var hours = d.getHours();
console.log(hours);
```

Let's create a program that prints the current time to the browser once every second.

```
function printTime() {
    var d = new Date();
    var hours = d.getHours();
    var mins = d.getMinutes();
    var secs = d.getSeconds();
    document.body.innerHTML = 
hours+":"+mins+":"+secs;
}
setInterval(printTime, 1000);
```

The Date Object

We declared a function printTime(), which gets the current time from the date object, and prints it to the screen.

We then called the function once every second, using the setInterval method.

The innerHTML property sets or returns the HTML content of an element.

In our case, we are changing the HTML content of our document's body.

This overwrites the content every second, instead of printing it repeatedly to the screen.

Task

The Date Object

The program you are given takes **year, month and day** as input.
Create a function that takes them as arguments and returns the corresponding **day of the week**.

Sample Input
1993
7
12

Sample Output
Thursday

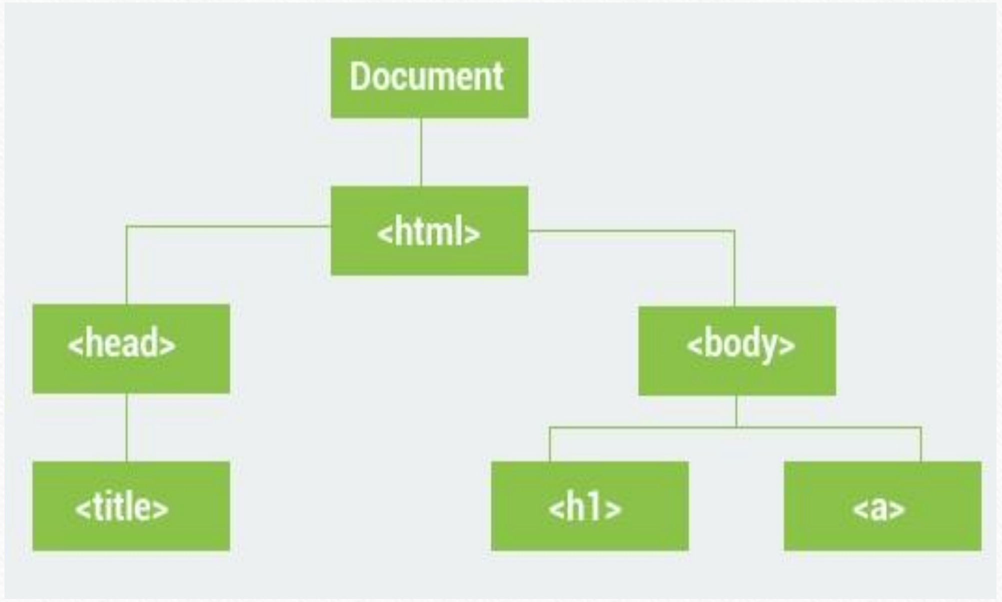
Hint: The given code creates a **Date** object from the parameters. Use the **getDay()** method of the date object to get the index, then use it in the given **names** array to return the name of the day.

What is DOM?

When you open any webpage in a browser, the HTML of the page is loaded and rendered visually on the screen.

To accomplish that, the browser builds the Document Object Model of that page, which is an object oriented model of its logical structure.

The DOM of an HTML document can be represented as a nested set of boxes:



What is DOM?

JavaScript can be used to manipulate the DOM of a page dynamically to add, delete and modify elements.

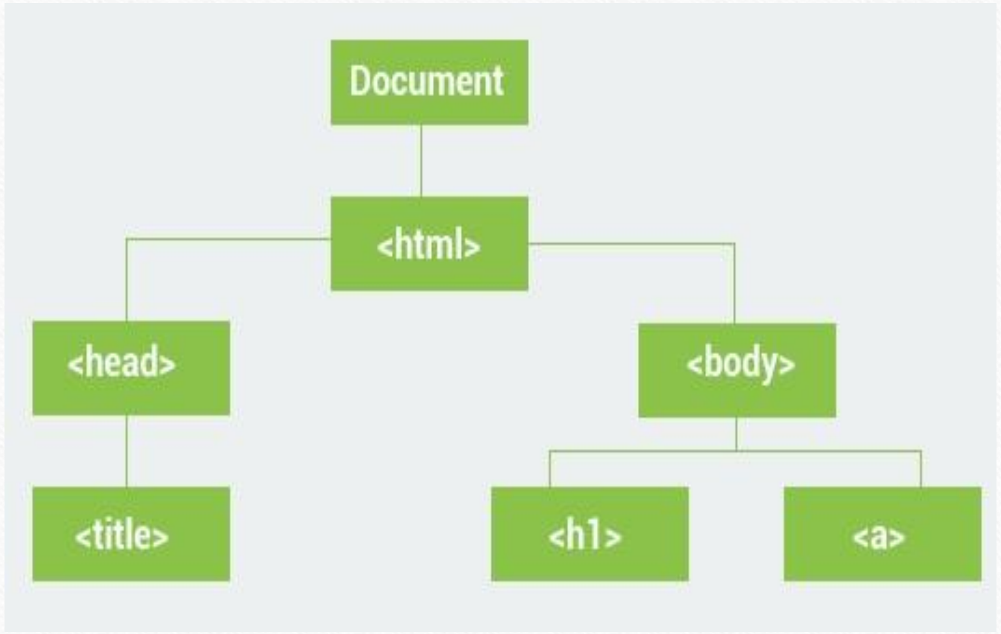
The DOM represents a document as a tree structure.

HTML elements become interrelated nodes in the tree.

All those nodes in the tree have some kind of relations among each other.

Nodes can have child nodes. Nodes on the same tree level are called siblings.

For example, consider the following structure:



What is DOM?

For the example above:

<html> has two children (<head>, <body>);

<head> has one child (<title>) and one parent (<html>);

<title> has one parent (<head>) and no children;

<body> has two children (<h1> and <a>) and one parent (<html>);

It is important to understand the relationships between elements in an HTML document in order to be able to manipulate them with JavaScript.

There is a predefined document object in JavaScript, which can be used to access all elements on the DOM.

In other words, the document object is the owner (or root) of all objects in your webpage.

So, if you want to access objects in an HTML page, you always start with accessing the document object.

For example:

```
document.body.innerHTML = "Some text";
```

As body is an element of the DOM, we can access it using the document object and change the content of the innerHTML property.

The innerHTML property can be used on almost all HTML elements to change its content.

Selecting Elements

All HTML elements are objects. And as we know every object has properties and methods.

The document object has methods that allow you to select the desired HTML element.

These three methods are the most commonly used for selecting HTML elements:

```
//finds element by id
document.getElementById(id)
//finds elements by class name
document.getElementsByClassName(name)
//finds elements by tag name
document.getElementsByTagName(name)
```

Selecting Elements

In the example below, the getElementById method is used to select the element with id="demo" and change its content:

```
var elem = document.getElementById("demo");
elem.innerHTML = "Hello World!";
```

The example above assumes that the HTML contains an element with id="demo", for example <div id="demo"></div>.

The getElementsByClassName() method returns a collection of all elements in the document with the specified class name.

For example, if our HTML page contained three elements with class="demo", the following code would return all those elements as an array:

```
var arr =  document.getElementsByClassName("demo");
//accessing the second element
arr[1].innerHTML = "Hi";
```

Selecting Elements

Similarly, the getElementsByTagName method returns all of the elements of the specified tag name as an array.

The following example gets all paragraph elements of the page and changes their content:

```
<p>hi</p>

<p>hello</p>

<p>hi</p>

<script>

var arr = document.getElementsByTagName("p");

for (var x = 0; x < arr.length; x++) {

  arr[x].innerHTML = "Hi there";

}

</script>
```

The script will result in the following HTML:

```
<p>Hi there</p>

<p>Hi there</p>

<p>Hi there</p>
```

We used the length property of the array to loop through all the selected elements in the above example.

Selecting Elements

Each element in the DOM has a set of properties and methods that provide information about their relationships in the DOM:

element.childNodes returns an array of an element's child nodes.

element.firstChild returns the first child node of an element.

element.lastChild returns the last child node of an element.

element.hasChildNodes returns true if an element has any child nodes, otherwise false.

element.nextSibling returns the next node at the same tree level.

element.previousSibling returns the previous node at the same tree level.

element.parentNode returns the parent node of an element.

Selecting Elements

We can, for example, select all child nodes of an element and change their content:

```
<html>
   <body>
      <div id ="demo">
         <p>some text</p>
         <p>some other text</p>
      </div>
   </body>
</html>
```

```
function setText() {
   var a = document.getElementById("demo");
   var arr = a.childNodes;
   for(var x=0;x<arr.length;x++) {
     arr[x].innerHTML = "new text";
   }
}
//calling the function with setTimeout to make sure the HTML is loaded
setTimeout(setText, 500);
```

The code above changes the text of both paragraphs to "new text".

Changing Elements

Once you have selected the element(s) you want to work with, you can change their attributes.

As we have seen in the previous lessons, we can change the text content of an element using the innerHTML property.

Similarly, we can change the attributes of elements.

For example, we can change the src attribute of an image:

```
<img id="myimg" src="orange.png" alt="" />

<script>

var el = document.getElementById("myimg");

el.src = "apple.png";

</script>
```

Changing Elements

We can change the href attribute of a link:

```
<!DOCTYPE html>
<html>
	<head>
		<title>Page Title</title>
	</head>
	<body>
		<a href="http://www.example.com">Some link</a>
	</body>
</html>
```

```
//calling the function in window.onload to make sure the HTML is loaded
window.onload = function() {
    var el = document.getElementsByTagName('a');
    el[0].href= 'http://www.google.com';
};
```

Practically all attributes of an element can be changed using JavaScript.

Changing Elements

The style of HTML elements can also be changed using JavaScript.

All style attributes can be accessed using the style object of the element.

For example:

```
<!DOCTYPE html>
<html>
        <head>
                <title>Page Title</title>
        </head>
        <body>
                <div id="demo" style="width:200px">some text</div>
        </body>
</html>
```

```
//calling the function in window.onload to make sure the HTML is loaded
window.onload = function() {
   var x = document.getElementById("demo");
   x.style.color = '#6600FF';
   x.style.width = '100px';
};
```

The code above changes the text color and width of the div element.

Changing Elements

All CSS properties can be set and modified using JavaScript.

Just remember, that you cannot use dashes (-) in the property names:

these are replaced with camelCase versions, where the compound words begin with a capital letter.

For example: the background-color property should be referred to as backgroundColor.

Adding & Removing Elements

Use the following methods to create new nodes:

element.cloneNode() clones an element and returns the resulting node.

document.createElement(element) creates a new element node.

document.createTextNode(text) creates a new text node.

For example:

var node = document.createTextNode("Some new text");

This will create a new text node,

but it will not appear in the document until you append it to an existing element with one of the following methods:

element.appendChild(newNode) adds a new child node to an element as the last child node.

element.insertBefore(node1, node2) inserts node1 as a child before node2.

Adding & Removing Elements

This creates a new paragraph and adds it to the existing div element on the page.

```
<!DOCTYPE html>
<html>
	<head>
		<title>Page Title</title>
	</head>
	<body>
		<div id="demo">some content</div>
	</body>
</html>
```

```
//calling the function in window.onload to make sure the HTML is loaded
window.onload = function() {
   //creating a new paragraph
   var p = document.createElement("p");
   var node = document.createTextNode("Some new text");
   //adding the text to the paragraph
   p.appendChild(node);

   var div = document.getElementById("demo");
   //adding the paragraph to the div
   div.appendChild(p);
};
```

Adding & Removing Elements

To remove an HTML element, you must select the parent of the element and use the removeChild(node) method.

```
<!DOCTYPE html>
<html>
        <head>
                <title>Page Title</title>
        </head>
        <body>
                <div id="demo">
       <p id="p1">This is a paragraph.</p>
       <p id="p2">This is another paragraph.</p>
     </div>
        </body>
</html>
```

```
//calling the function in window.onload to make sure the HTML is loaded
window.onload = function() {
   var parent = document.getElementById("demo");
   var child = document.getElementById("p1");
   parent.removeChild(child);
};
```

This removes the paragraph with id="p1" from the page.

Adding & Removing Elements

An alternative way of achieving the same result would be the use of the parentNode property to get the parent of the element we want to remove:

```
var child = document.getElementById("p1");
```

```
child.parentNode.removeChild(child);
```

To replace an HTML element, the element.replaceChild(newNode, oldNode) method is used.

For example:

Adding & Removing Elements

```
<!DOCTYPE html>
<html>
	<head>
		<title>Page Title</title>
	</head>
	<body>
		<div id="demo">
	 <p id="p1">This is a paragraph.</p>
	 <p id="p2">This is another paragraph.</p>
	</div>
	</body>
</html>
```

```
//calling the function in window.onload to make sure the HTML is loaded
window.onload = function() {
   var p = document.createElement("p");
   var node = document.createTextNode("This is new");
   p.appendChild(node);
   var parent = document.getElementById("demo");
   var child = document.getElementById("p1");
   parent.replaceChild(p, child);
};
```

The code above creates a new paragraph element that replaces the existing p1 paragraph.

Creating Animations

Now that we know how to select and change DOM elements, we can create a simple animation.

Let's create a simple HTML page with a box element that will be animated using JS.

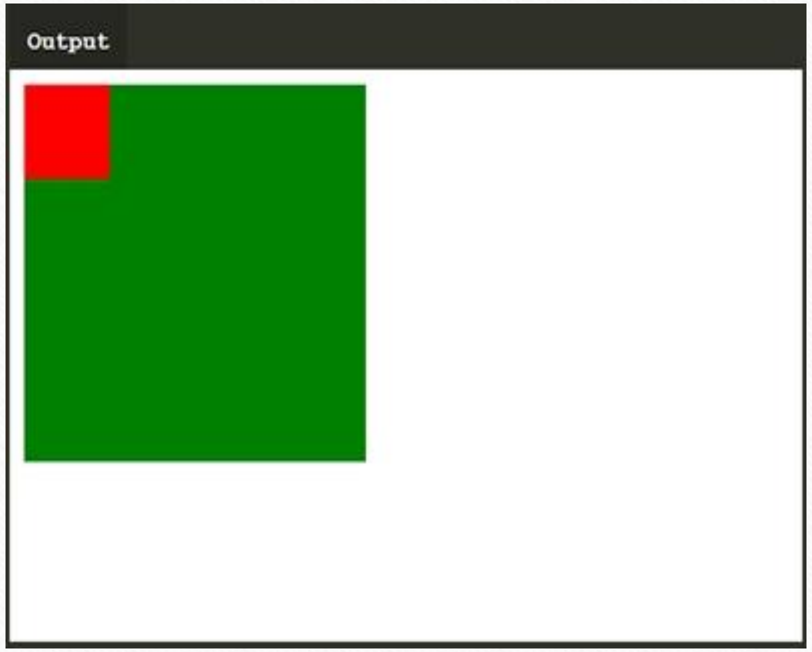
```
<!DOCTYPE html>
<html>
        <head>
                <title>Page Title</title>
        </head>
        <body>
                <div id="container">
        <div id="box"> </div>
     </div>
        </body>
</html>
```

```
#container {
  width: 200px;
  height: 200px;
  background: green;
  position: relative;
}
#box {
  width: 50px;
  height: 50px;
  background: red;
  position: absolute;
}
```

Creating Animations

Our box element is inside a container element. Note the position attribute used for the elements:

the container is relative and the box is absolute. This will allow us to create the animation relative to the container.



Creating Animations

We will be animating the red box to make it move to the right side of the container.

You need to be familiar with CSS to better understand the code provided.

To create an animation, we need to change the properties of an element at small intervals of time.

We can achieve this by using the setInterval() method,

which allows us to create a timer and call a function to change properties repeatedly at defined intervals (in milliseconds).

For example:

```
var t = setInterval(move, 500);
```

This code creates a timer that calls a move() function every 500 milliseconds.

Now we need to define the move() function, that changes the position of the box.

Creating Animations

```
// starting position
var pos = 0;
//our box element
var box = document.getElementById("box");
function move() {
  pos += 1;
  box.style.left = pos+"px"; //px = pixels
}
```

The move() function increments the left property of the box element by one each time it is called.

The following code defines a timer that calls the move() function every 10 milliseconds:

```
var t = setInterval(move, 10);
```

However, this makes our box move to the right forever. To stop the animation when the box reaches the end of the container, we add a simple check to the move() function and use the clearInterval() method to stop the timer.

Creating Animations

```
function move() {
 if(pos >= 150) {
   clearInterval(t);
 }
 else {
   pos += 1;
   box.style.left = pos+"px";
 }
}
```

When the left attribute of the box reaches the value of 150, the box reaches the end of the container, based on a container width of 200 and a box width of 50.

Creating Animations

The final code:

```
<!DOCTYPE html>
<html>
      <head>
            <title>Page Title</title>
      </head>
      <body>
            <div id="container">
      <div id="box"> </div>
    </div>
      </body>
</html>
```

```
#container {
  width: 200px;
  height: 200px;
  background: green;
  position: relative;
}
#box {
  width: 50px;
  height: 50px;
  background: red;
  position: absolute;
}
```

Creating Animations

```
//calling the function in window.onload to make sure the HTML is loaded

window.onload = function() {
    var pos = 0;
    //our box element
    var box = document.getElementById('box');
    var t = setInterval(move, 10);

function move() {
        if(pos >= 150) {
            clearInterval(t);
        }
        else {
            pos += 1;
            box.style.left = pos+'px';
        }
    }
};
```

Congratulations, you have just created your first JavaScript animation!

Handling Events

You can write JavaScript code that executes when an event occurs, such as when a user clicks an HTML element,

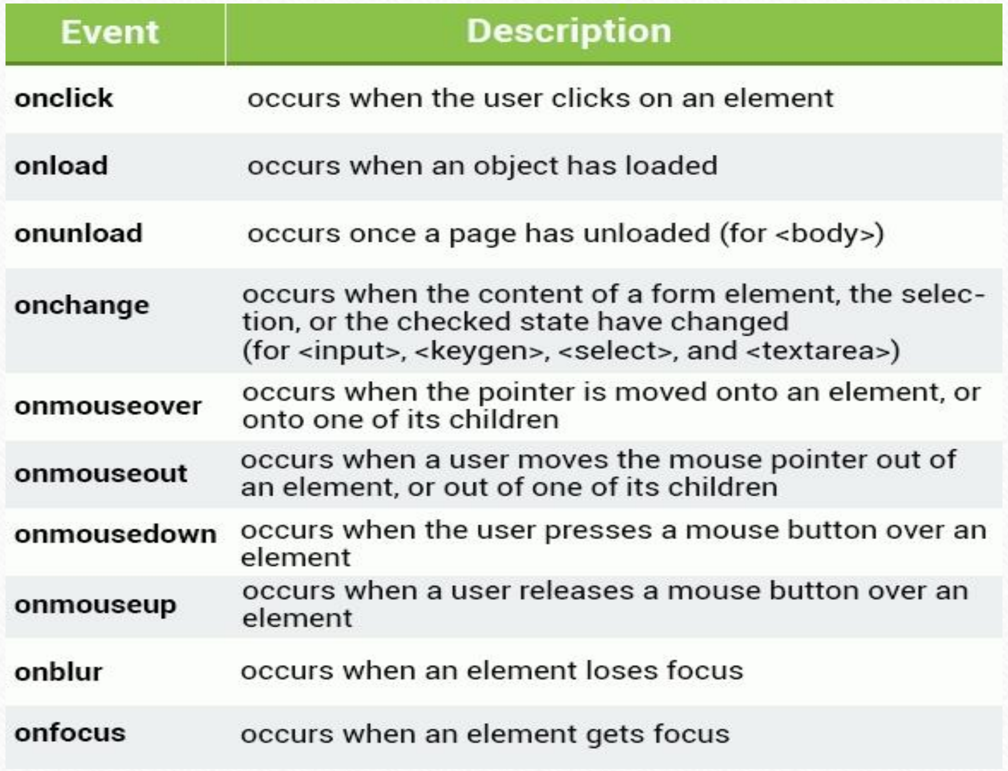
moves the mouse, or submits a form.

When an event occurs on a target element, a handler function is executed.

Common HTML events include:

Corresponding events can be added to HTML elements as attributes.

For example: <p onclick="someFunc()">some text</p>



Event	Description
onclick	occurs when the user clicks on an element
onload	occurs when an object has loaded
onunload	occurs once a page has unloaded (for <body>)
onchange	occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <keygen>, <select>, and <textarea>)
onmouseover	occurs when the pointer is moved onto an element, or onto one of its children
onmouseout	occurs when a user moves the mouse pointer out of an element, or out of one of its children
onmousedown	occurs when the user presses a mouse button over an element
onmouseup	occurs when a user releases a mouse button over an element
onblur	occurs when an element loses focus
onfocus	occurs when an element gets focus

Handling Events

Let's display an alert popup when the user clicks a specified button:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <button onclick="show();">Click Me</button>
    </body>
</html>
```

```
function show() {
   alert("Hi there");
}
```

Handling Events

Event handlers can be assigned to elements.

For example:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <button id="demo">Click Me</button>
    </body>
</html>
```

```
//calling the function in window.onload to make sure the HTML is loaded
window.onload = function() {
   var x = document.getElementById('demo');
   x.onclick = function () {
      document.body.innerHTML = Date();
   }
};
```

You can attach events to almost all HTML elements.

Handling Events

The onload and onunload events are triggered when the user enters or leaves the page.

These can be useful when performing actions after the page is loaded.

```
<body onload="doSomething()">
```

Similarly, the window.onload event can be used to run code after the whole page is loaded.

```
window.onload = function() {
   //some code
}
```

Handling Events

The onchange event is mostly used on textboxes.

The event handler gets called when the text inside the textbox changes and focus is lost from the element.

For example:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <input type="text" id="name" onchange="change()">
    </body>
</html>
```

```
function change() {
  var x = document.getElementById('name');
  x.value = x.value.toUpperCase();
}
```

It’s important to understand events, because they are an essential part of dynamic web pages.

Handling Events

The addEventListener() method attaches an event handler to an element without overwriting existing event handlers.

You can add many event handlers to one element.

You can also add many event handlers of the same type to one element, i.e., two "click" events.

element.addEventListener(event, function, useCapture);

The first parameter is the event's type (like "click" or "mousedown").

The second parameter is the function we want to call when the event occurs.

The third parameter is a Boolean value specifying whether to use event bubbling or event capturing.

This parameter is optional, and will be described in the next lesson.

Note that you don't use the "on" prefix for this event; use "click" instead of "onclick".

Handling Events

Example:

```
element.addEventListener("click", myFunction);
element.addEventListener("mouseover", myFunction);
function myFunction() {
  alert("Hello World!");
}
```

This adds two event listeners to the element.

We can remove one of the listeners:

```
element.removeEventListener("mouseover", myFunction);
```

Handling Events

Let's create an event handler that removes itself after being executed:

```
<!DOCTYPE html>
<html>
	<head>
		<title>Page Title</title>
	</head>
	<body>
		<button id="demo">Start</button>
	</body>
</html>
```

```
//calling the function in window.onload to make sure the HTML is loaded
window.onload = function() {
    var btn = document.getElementById("demo");
    btn.addEventListener("click", myFunction);

    function myFunction() {
        alert(Math.random());
        btn.removeEventListener("click", myFunction);
    }
};
```

Handling Events

After clicking the button, an alert with a random number displays and the event listener is removed.

Internet Explorer version 8 and lower do not support the addEventListener() and removeEventListener() methods.

However, you can use the document.attachEvent() method to attach event handlers in Internet Explorer.

Event Propagation

There are two ways of event propagation in the HTML DOM: bubbling and capturing.

Event propagation allows for the definition of the element order when an event occurs.

If you have a <p> element inside a <div> element, and the user clicks on the <p> element, which element's "click" event should be handled first?

In bubbling, the innermost element's event is handled first and then the outer element's event is handled.

The <p> element's click event is handled first, followed by the <div> element's click event.

In capturing, the outermost element's event is handled first and then the inner.

The <div> element's click event is handled first, followed by the <p> element's click event.

Capturing goes down the DOM.

Bubbling goes up the DOM.

Event Propagation

The addEventListener() method allows you to specify the propagation type with the "useCapture" parameter.

```
addEventListener(event, function, useCapture)
```

The default value is false, which means the bubbling propagation is used;

when the value is set to true, the event uses the capturing propagation.

```
//Capturing propagation
elem1.addEventListener("click", myFunction, true);
//Bubbling propagation
elem2.addEventListener("click", myFunction, false);
```

This is particularly useful when you have the same event handled for multiple elements in the DOM hierarchy.

Creating an Image Slider

Now we can create a sample image slider project. The images will be changed using "Next" and "Prev" buttons.

Now, let’s create our HTML, which includes an image and the two navigation buttons:

```
<!DOCTYPE html>
<html>
        <head>
                <title>Page Title</title>
        </head>
        <body>
    <div>
          <button> Prev </button>
          <img src="http://www.google.com/uploads/slider/1.jpg" width="200px" height="100px" />
      <button> Next </button>
    </div>
        </body>
</html>
```

```
button {
  margin-top:30px;
  float:left;
  height:50px;
}
img {
  float:left;
  margin-right:10px;
  margin-left:10px;
 }
```

Next, let's define our sample images in an array:

```
var images = [
  "http://www.google.com/uploads/slider/1.jpg",
  "http://www.google.com/uploads/slider/2.jpg",
  "http://www.google.com/uploads/slider/3.jpg"
];
```

Creating an Image Slider

We are going to use three sample images that we have uploaded to our server. You can use any number of images.

Now we need to handle the Next and Prev button clicks and call the corresponding functions to change the image.

HTML:

```
<div>
  <button onclick="prev()"> Prev </button>
  <img id="slider" src="http://www.google.com/uploads/slider/1.jpg"
    width="200px" height="100px"/>
  <button onclick="next()"> Next </button>
</div>
```

Creating an Image Slider

```
<!DOCTYPE html>
<html>
        <head>
                <title>Page Title</title>
        </head>
        <body>
    <div>
      <button onclick="prev()"> Prev </button>
      <img id="slider" src="http://www.google.com/uploads/slider/1.jpg" width="200px"
height="100px"/>
      <button onclick="next()"> Next </button>
    </div>
        </body>
</html>
```

```
button {
  margin-top:30px;
  float:left;
  height:50px;
}
img {
  float:left;
  margin-right:10px;
  margin-left:10px;
}
var images = [
  'http://www.google.com/uploads/slider/1.jpg',
  'http://www.google.com/uploads/slider/2.jpg',
  'http://www.google.com/uploads/slider/3.jpg'
];
```

Creating an Image Slider

```
var num = 0;
function next() {
    var slider = document.getElementById('slider');
    num++;
    if(num >= images.length) {
        num = 0;
    }
    slider.src = images[num];
}
```

```
function prev() {
    var slider = document.getElementById('slider');
    num--;
    if(num < 0) {
        num = images.length-1;
    }
    slider.src = images[num];
}
```

Creating an Image Slider

The num variable holds the current image.

The next and previous button clicks are handled by their corresponding functions,

which change the source of the image to the next/previous image in the array.

We have created a functioning image slider!

Form Validation

HTML5 adds some attributes that allow form validation.

For example, the required attribute can be added to an input field to make it mandatory to fill in.

More complex form validation can be done using JavaScript.

The form element has an onsubmit event that can be handled to perform validation.

For example, let's create a form with two inputs and one button.

The text in both fields should be the same and not blank to pass the validation.

```
<form onsubmit="return validate()" method="post">
  Number: <input type="text" name="num1" id="num1" />
  <br />
  Repeat: <input type="text" name="num2" id="num2" />
  <br />
  <input type="submit" value="Submit" />
</form>
```

Form Validation

Now we need to define the validate() function:

```
<form onsubmit="return validate()" method="post">
   Number: <input type="text" name="num1" id="num1" /><br />
   Repeat: <input type="text" name="num2" id="num2" /><br />
   <input type="submit" value="Submit" />
</form>
```

```
function validate() {
   var n1 = document.getElementById('num1');
   var n2 = document.getElementById('num2');
   if(n1.value != '' && n2.value != '') {
      if(n1.value == n2.value) {
         return true;
      }
   }
   alert("The values should be equal and not blank");
   return false;
}
```

We return true only when the values are not blank and are equal.

The form will not get submitted if its onsubmit event returns false.

Intro to ES6

ECMAScript (ES) is a scripting language specification created to standardize JavaScript.

The Sixth Edition, initially known as ECMAScript 6 (ES6) and later renamed to ECMAScript 2015,

adds significant new syntax for writing complex applications, including classes and modules,

iterators and for/of loops, generators, arrow functions, binary data, typed arrays, collections (maps, sets and weak maps), promises,

number and math enhancements, reflection, and proxies.

In other words, ES6 is a superset of JavaScript (ES5).

The reason that ES6 became so popular is that it introduced new conventions and OOP concepts such as classes.

In this module, we cover the most important additions to ES6.

So, let's jump right in!

ES6 Variables and Strings

In ES6 we have three ways of declaring variables:

```
var a = 10;

const b = 'hello';

let c = true;
```

The type of declaration used depends on the necessary scope.

Scope is the fundamental concept in all programming languages that defines the visibility of a variable.

ES6 Variables and Strings

var & let

Unlike the var keyword, which defines a variable globally, or locally to an entire function regardless of block scope,

let allows you to declare variables that are limited in scope to the block, statement, or expression in which they are used.

For example:

```
if (true) {
    let name = 'Jack';
}
alert(name);
```

In this case, the name variable is accessible only in the scope of the if statement because it was declared as let.

Form Validation

To demonstrate the difference in scope between var and let, consider this example:

```
function varTest() {
    var x = 1;
    if (true) {
        var x = 2;  // same variable
        console.log(x);  // 2
    }
    console.log(x);  // 2
}
```

```
function letTest() {
    let x = 1;
    if (true) {
        let x = 2;  // different variable
        console.log(x);  // 2
    }
    console.log(x);  // 1
}
varTest();
letTest();
```

ES6 Variables and Strings

One of the best uses for let is in loops:

```
for (let i = 0; i < 3; i++) {
  document.write(i);
}
```

Here, the i variable is accessible only within the scope of the for loop, where it is needed.

let is not subject to Variable Hoisting, which means that let declarations do not move to the top of the current execution context.

const variables have the same scope as variables declared using let.

The difference is that const variables are immutable - they are not allowed to be reassigned.

For example, the following generates an exception:

```
const a = 'Hello';
a = 'Bye';
```

ES6 Variables and Strings

const is not subject to Variable Hoisting too, which means that const declarations do not move to the top of the current execution context.

Also note that ES6 code will run only in browsers that support it.

Older devices and browsers that do not support ES6 will return a syntax error.

Template literals are a way to output variables in the string.

Prior to ES6 we had to break the string, for example:

```
let name = 'David';

let msg = 'Welcome ' + name + '!';

console.log(msg);
```

ES6 Variables and Strings

ES6 introduces a new way of outputting variable values in strings. The same code above can be rewritten as:

```
let name = 'David';
let msg = `Welcome ${name}!`;
console.log(msg);
```

Notice, that template literals are enclosed by the backtick (` `) character instead of double or single quotes.

The ${expression} is a placeholder, and can include any expression, which will get evaluated and inserted into the template literal.

For example:

```
let a = 8;
let b = 34;
let msg = `The sum is ${a+b}`;
console.log(msg);
```

To escape a backtick in a template literal, put a backslash \ before the backtick.

Task

ES6 Variables and Strings

You need to make country cards for a school project.
The given program takes the country and its capital name as input.
Complete the function to return a string in the format you are given in the sample output:

Sample Input
Portugal
Lisbon

Sample Output
Name: Portugal, Capital: Lisbon

Use template literals to output variables in strings.

Loops and Functions in ES6

In JavaScript we commonly use the for loop to iterate over values in a list:

```
let arr = [1, 2, 3];
for (let k = 0; k < arr.length; k++) {
   console.log(arr[k]);
}
```

The for...in loop is intended for iterating over the enumerable keys of an object.

For example:

```
let obj = {a: 1, b: 2, c: 3};
for (let v in obj) {
   console.log(v);
}
```

Loops and Functions in ES6

The for...in loop should NOT be used to iterate over arrays because, depending on the JavaScript engine, it could iterate in an arbitrary order. Also, the iterating variable is a string, not a number, so if you try to do any math with the variable, you'll be performing string concatenation instead of addition.

ES6 introduces the new for...of loop, which creates a loop iterating over iterable objects.

For example:

```
let list = ["x", "y", "z"];
for (let val of list) {
   console.log(val);
}
```

During each iteration the val variable is assigned the corresponding element in the list.

Loops and Functions in ES6

The for...of loop works for other iterable objects as well, including strings

```
for (let ch of "Hello") {
    console.log(ch);
}
```

The for...of loop also works on the newly introduced collections (Map, Set, WeakMap, and WeakSet).

We will learn about them in the upcoming lessons.

Note that ES6 code will run only in browsers that support it. Older devices and browsers that do not support ES6 will return a syntax error.

Loops and Functions in ES6

Prior to ES6, a JavaScript function was defined like this:

```
function add(x, y) {
   var sum = x+y;
   console.log(sum);
}
add(35, 7);
```

ES6 introduces a new syntax for writing functions. The same function from above can be written as:

```
const add = (x, y) => {
   let sum = x + y;
   console.log(sum);
}
add(35, 7);
```

Loops and Functions in ES6

This new syntax is quite handy when you just need a simple function with one argument.

You can skip typing function and return, as well as some parentheses and braces.

For example:

```
const greet = x => "Welcome " + x;

alert(greet("David"));
```

The code above defines a function named greet that has one argument and returns a message.

Loops and Functions in ES6

If there are no parameters, an empty pair of parentheses should be used, as in

```
const x = () => alert("Hi");
x();
```

The syntax is very useful for inline functions. For example, let's say we have an array,

and for each element of the array we need to execute a function. We use the forEach method of the array to call a function for each element:

```
var arr = [2, 3, 7, 8];
arr.forEach(function(el) {
   console.log(el * 2);
});
```

Loops and Functions in ES6

However, in ES6, the code above can be rewritten as following:

```
const arr = [2, 3, 7, 8];
arr.forEach(v => {
    console.log(v * 2);
});
```

The code is shorter and looks pretty nice, doesn't it?

Loops and Functions in ES6

In ES6, we can put the default values right in the signature of the functions.

For example:

```
/*
function test(a, b = 3, c = 42) {
  return a + b + c;
}
console.log(test(5));
*/
// Full ES6 equivalent
const test = (a, b = 3, c = 42) => a + b + c;
console.log(test(5));
```

Loops and Functions in ES6

And here's an example of an arrow function with default parameters:

```
const test = (a, b = 3, c = 42) => {
 return a + b + c;
}
console.log(test(5)); //50
```

Default value expressions are evaluated at function call time from left to right.

This also means that default expressions can use the values of previously-filled parameters.

Task

Loops in ECMAScript 6

Students need to score at least 70 points to pass an exam. The given program declares an array with results.
Write a program to count and output to the console the number of students who pass the exam.

Use a for...of loop to iterate through the array.

ES6 Objects

JavaScript variables can be Object data types that contain many values called properties.

An object can also have properties that are function definitions called methods for performing actions on the object.

ES6 introduces shorthand notations and computed property names that make declaring and using objects easier to understand.

The new method definition shorthand does not require the colon (:) or function keyword, as in the grow function of the tree object declaration:

```
let tree = {
  height: 10,
  color: 'green',
  grow() {
    this.height += 2;
  }
};
tree.grow();
console.log(tree.height); // 12
```

ES6 Objects

You can also use a property value shorthand when initializing properties with a variable by the same name.

For example, properties height and health are being initialized with variables named height and health

```
let height = 5;
let health = 100;
let athlete = {
    height, // height: height,
    health // health: health
};
console.log(athlete.height); // 5
```

ES6 Objects

When creating an object by using duplicate property names, the last property will overwrite the prior ones of the same name.

For Example:

```
var a = {x: 1, x: 2, x: 3, x: 4};
console.log(a.x); // 4
```

Duplicate property names generated a SyntaxError in ES5 when using strict mode. However, ES6 removed this limitation.

With ES6, you can now use computed property names. Using the square bracket notation [], we can use an expression for a property name, including concatenating strings. This can be useful in cases where we want to create certain objects based on user data (e.g. id, email, and so on).

ES6 Objects

Here are three examples:

Example 1:

```
let prop = 'name';
let id = '1234';
let mobile = '08923';
let user = {
  [prop]: 'Jack',
  [`user_${id}`]: `${mobile}`
};
console.log(user.name); // Jack
console.log(user.user_1234); // 08923
```

ES6 Objects

Example 2:

```
var i = 0;
var a = {
  ['foo' + ++i]: i,
  ['foo' + ++i]: i,
  ['foo' + ++i]: i
};
console.log(a.foo1); // 1
console.log(a.foo2); // 2
console.log(a.foo3); // 3
```

ES6 Objects

Example 3:

```
var param = 'size';
var config = {
    [param]: 12,
    ['mobile' + param.charAt(0).toUpperCase() + param.slice(1)]: 4
};
console.log(config.mobileSize); // 4
```

It is very useful when you need to create custom objects based on some variables.

ES6 adds a new Object method assign() that allows us to combine multiple sources into one target to create a single new object.

Object.assign() is also useful for creating a duplicate of an existing object.

ES6 Objects

Let's look at the following example to see how to combine objects:

```
let person = {
    name: 'Jack',
    age: 18,
    sex: 'male'
};
let student = {
    name: 'Bob',
    age: 20,
    xp: '2'
};
```

```
let newStudent = Object.assign({}, person, student);
console.log(newStudent.name); // Bob
console.log(newStudent.age); // 20
console.log(newStudent.sex); // male
console.log(newStudent.xp); // 2
```

ES6 Objects

Here we used Object.assign() where the first parameter is the target object you want to apply new properties to.

Every parameter after the first will be used as sources for the target. There are no limitations on the number of source parameters.

However, order is important because properties in the second parameter will be overridden by properties of the same name in third parameter,

and so on.

In the example above, we used a new object {} as the target and used two objects as sources.

Try changing the order of second and third parameters to see what happens to the result.

ES6 Objects

Now, let's see how we can use assign() to create a duplicate object without creating a reference (mutating) to the base object.

In the following example, assignment was used to try to generate a new object. However, using = creates a reference to the base object.

Because of this reference, changes intended for a new object mutate the original object:

```
let person = {
   name: 'Jack',
   age: 18
};
let newPerson = person;
newPerson.name = 'Bob';
console.log(person.name); // Bob
console.log(newPerson.name); // Bob
```

ES6 Objects

To avoid this (mutations), use Object.assign() to create a new object.

For example:

```
let person = {
    name: 'Jack',
    age: 18
};
let newPerson = Object.assign({}, person);
newPerson.name = 'Bob';
console.log(person.name); // Jack
console.log(newPerson.name); // Bob
```

ES6 Objects

Finally, you can assign a value to an object property in the Object.assign() statement.

For example:

```
let person = {
    name: 'Jack',
    age: 18
};
let newPerson = Object.assign({}, person, {name: 'Bob'});
console.log(newPerson.name); // Bob
```

Run the code and see how it works!

Task

ES6 Objects

You are making a workout app. After completing the basic exercises in the app, the user is able to access advanced content.
The given program declares two classes - basic and advanced with corresponding properties. Complete the code to combine basic and advanced level exercises into one new object named **total**, so that the given code for final output works correctly.

Use **Object.assign()** to perform the requested operation.

ES6 Destructuring

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays,

or properties from objects, into distinct variables.

ES6 has added a shorthand syntax for destructuring an array.

The following example demonstrates how to unpack the elements of an array into distinct variables:

```
let arr = ['1', '2', '3'];
let [one, two, three] = arr;
console.log(one); // 1
console.log(two); // 2
console.log(three); // 3
```

ES6 Destructuring

We can use also destructure an array returned by a function.

For example:

```
let a = () => {
    return [1, 3, 2];
};
let [one, , two] = a();
console.log(one); // 1
console.log(two); // 2
```

Notice that we left the second argument's place empty.

ES6 Destructuring

The destructuring syntax also simplifies assignment and swapping values:

```
let a, b, c = 4, d = 8;
[a, b = 6] = [2];
console.log(a); // 2
console.log(b); // 6
[c, d] = [d, c];
console.log(c); // 8
console.log(d); // 4
```

Run the code and see how it works!

ES6 Destructuring

Similar to Array destructuring, Object destructuring unpacks properties into distinct variables.

For example:

```
let obj = {h:100, s: true};

let {h, s} = obj;

console.log(h);

console.log(s);
```

ES6 Destructuring

We can assign without declaration, but there are some syntax requirements for that:

```
let a, b;
({a, b} = {a: 'Hello ', b: 'Jack'});
console.log(a + b);
```

The () with a semicolon (;) at the end are mandatory when destructuring without a declaration.

However, you can also do it as follows where the () are not required:

```
let {a, b} = {a: 'Hello ', b: 'Jack'};
console.log(a + b);
```

ES6 Destructuring

You can also assign the object to new variable names.

For example:

```
var o = {h: 42, s: true};
var {h: foo, s: bar} = o;
//console.log(h); // Error
console.log(foo); // 42
```

ES6 Destructuring

Finally you can assign default values to variables, in case the value unpacked from the object is undefined.

For example:

```
var obj = {id: 42, name: "Jack"};
let {id = 10, age = 20} = obj;
console.log(id); // 42
console.log(age); // 20
```

Run the code and see how it works!

Rest & Spread

Prior to ES6, if we wanted to pass a variable number of arguments to a function, we could use the arguments object, an array-like object, to access the parameters passed to the function.

For example, let's write a function that checks if an array contains all the arguments passed:

```
function containsAll(arr) {
    for (let k = 1; k < arguments.length; k++) {
        let num = arguments[k];
        if (arr.indexOf(num) === -1) {
            return false;
        }
    }
    return true;
}
let x = [2, 4, 6, 7];
console.log(containsAll(x, 2, 4, 7));
console.log(containsAll(x, 6, 4, 9));
```

Rest & Spread

We can pass any number of arguments to the function and access it using the arguments object.

While this does the job, ES6 provides a more readable syntax to achieve variable number of parameters by using a rest parameter:

```
function containsAll(arr, ...nums) {
    for (let num of nums) {
        if (arr.indexOf(num) === -1) {
            return false;
        }
    }
    return true;
}
let x = [2, 4, 6, 7];
console.log(containsAll(x, 2, 4, 7));
console.log(containsAll(x, 6, 4, 9));
```

Rest & Spread

The ...nums parameter is called a rest parameter. It takes all the "extra" arguments passed to the function.

The three dots (...) are called the Spread operator.

Only the last parameter of a function may be marked as a rest parameter.

If there are no extra arguments, the rest parameter will simply be an empty array; the rest parameter will never be undefined.

This operator is similar to the Rest Parameter, but it has another purpose when used in objects or arrays or function calls (arguments).

Rest & Spread

Spread in function calls

It is common to pass the elements of an array as arguments to a function. Before ES6, we used the following method:

```
function myFunction(w, x, y, z) {
    console.log(w + x + y + z);
}
var args = [1, 2, 3];
myFunction.apply(null, args.concat(4));
```

ES6 provides an easy way to do the example above with spread operators

```
const myFunction = (w, x, y, z) => {
    console.log(w + x + y + z);
};
let args = [1, 2, 3];
myFunction(...args, 4);
```

Rest & Spread

Example:

```
var dateFields = [1970, 0, 1];  // 1 Jan 1970
var date = new Date(...dateFields);
console.log(date);
```

Rest & Spread

Spread in array literals

Before ES6, we used the following syntax to add an item at middle of an array:

```
var arr = ["One", "Two", "Five"];
arr.splice(2, 0, "Three");
arr.splice(3, 0, "Four");
console.log(arr);
```

You can use methods such as push, splice, and concat, for example, to achieve this in different positions of the array.

However, in ES6 the spread operator lets us do this more easily:

```
let newArr = ['Three', 'Four'];
let arr = ['One', 'Two', ...newArr, 'Five'];
console.log(arr);
```

Rest & Spread

Spread in object literals

In objects it copies the own enumerable properties from the provided object onto a new object.

```
const obj1 = { foo: 'bar', x: 42 };

const obj2 = { foo: 'baz', y: 5 };

const clonedObj = { ...obj1 }; // { foo: "bar", x: 42 }

const mergedObj = { ...obj1, ...obj2 }; // { foo: "baz", x: 42, y: 5 }
```

Rest & Spread

However, if you try to merge them you will not get the result you expected:

```
const obj1 = { foo: 'bar', x: 42 };
const obj2 = { foo: 'baz', y: 5 };
const merge = (...objects) => ({ ...objects });
let mergedObj = merge (obj1, obj2);
// { 0: { foo: 'bar', x: 42 }, 1: { foo: 'baz', y: 5 } }
let mergedObj2 = merge ({}, obj1, obj2);
// { 0: {}, 1: { foo: 'bar', x: 42 }, 2: { foo: 'baz', y: 5 } }
```

Shallow cloning or merging objects is possible with another operator called Object.assign().

Task

Rest & Spread

You are making a program to calculate the sum of any number of values. Complete the given function so that it takes as parameters as many numbers as needed and returns the sum.

Use the **rest parameter** as an argument.

ES6 Classes

In this lesson we'll explain how to create a class that can be used to create multiple objects of the same structure.

A class uses the keyword class and contains a constructor method for initializing.

For example:

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

ES6 Classes

A declared class can then be used to create multiple objects using the keyword new.

For example:

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
 }
}
const square = new Rectangle(5, 5);
const poster = new Rectangle(2, 3);
console.log(square.height);
```

Class Declarations are not hoisted while Function Declarations are.

If you try to access your class before declaring it, ReferenceError will be returned.

ES6 Classes

You can also define a class with a class expression, where the class can be named or unnamed.

A named class looks like:

```
var Square = class Rectangle {
   constructor(height, width) {
      this.height = height;
      this.width = width;
  }
};
const square = new Square(5, 5);
const poster = new Square(2, 3);
console.log(square.height);
```

ES6 Classes

In the unnamed class expression, a variable is simply assigned the class definition:

```
var Square = class {
   constructor(height, width) {
      this.height = height;
      this.width = width;
  }
};
const square = new Square(5, 5);
const poster = new Square(2, 3);
console.log(square.height);
```

The constructor is a special method which is used for creating and initializing an object created with a class.

There can be only one constructor in each class.

ES6 Classes

ES6 introduced a shorthand that does not require the keyword function for a function assigned to a method's name.

One type of class method is the prototype method, which is available to objects of the class.

For Example:

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
 get area() {
    return this.calcArea();
  }
  calcArea() {
    return this.height * this.width;
  }
}
const square = new Rectangle(5, 5);
console.log(square.area); // 25
```

In the code above, area is a getter, calcArea is a method.

ES6 Classes

Another type of method is the static method, which cannot be called through a class instance.

Static methods are often used to create utility functions for an application.

For Example:

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
   static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;
    return Math.hypot(dx, dy);
  }
}
const p1 = new Point(7, 2);
const p2 = new Point(3, 8);
console.log(Point.distance(p1, p2));
```

As you can see, the static distance method is called directly using the class name, without an object.

ES6 Classes

The extends keyword is used in class declarations or class expressions to create a child of a class.

The child inherits the properties and methods of the parent.

For example:

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' makes a noise.');
  }
}
```

```
class Dog extends Animal {
  speak() {
    console.log(this.name + ' barks.');
  }
}
let dog = new Dog('Rex');
dog.speak(); // Rex barks.
```

In the code above, the Dog class is a child of the Animal class, inheriting its properties and methods.

ES6 Classes

If there is a constructor present in the subclass, it needs to first call super() before using this.

Also, the super keyword is used to call parent's methods.

For example, we can modify the program above to the following:

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' makes a noise.');
  }
}
class Dog extends Animal {
  speak() {
    super.speak(); // Super
    console.log(this.name + ' barks.');
  }
}
let dog = new Dog('Rex');
dog.speak();
// Rex makes a noise.
// Rex barks.
```

In the code above, the parent's speak() method is called using the super keyword.

Task

Class Methods in ES6

You are making a program so that students are able to calculate their average of 3 exam scores.
The given program takes the scores of 3 exams as input and declares the Exams class.
Add a **static method** average() to class Exams, which will take the scores as parameters, and calculate and return the average score **rounded to the nearest integer** so that the code in main works correctly.

Sample Input
74
80
68

Sample Output
74

Recall **Math.round()** function to round the number with the floating point to the nearest integer.

ES6 Map & Set

A Map object can be used to hold key/value pairs. A key or value in a map can be anything (objects and primitive values).

The syntax new Map([iterable]) creates a Map object where iterable is an array or any other iterable object whose elements are arrays (with a key/value pair each).

An Object is similar to Map but there are important differences that make using a Map preferable in certain cases:

- The keys can be any type including functions, objects, and any primitive.
- You can get the size of a Map.
- You can directly iterate over Map.
- Performance of the Map is better in scenarios involving frequent addition and removal of key/value pairs.

The size property returns the number of key/value pairs in a map.

For example:

```
let map = new Map([['k1', 'v1'], ['k2', 'v2']]);
console.log(map.size); // 2
```

ES6 Map & Set

Methods

- set(key, value) Adds a specified key/value pair to the map. If the specified key already exists,

value corresponding to it is replaced with the specified value.

- get(key) Gets the value corresponding to a specified key in the map. If the specified key doesn't exist, undefined is returned.
- has(key) Returns true if a specified key exists in the map and false otherwise.
- delete(key) Deletes the key/value pair with a specified key from the map and returns true. Returns false if the element does not exist.
- clear() Removes all key/value pairs from map.
- keys() Returns an Iterator of keys in the map for each element.
- values() Returns an Iterator of values in the map for each element.
- entries() Returns an Iterator of array[key, value] in the map for each element.

ES6 Map & Set

For example:

```
let map = new Map();
map.set('k1', 'v1').set('k2', 'v2');
console.log(map.get('k1')); // v1
console.log(map.has('k2')); // true
for (let kv of map.entries())
    console.log(kv[0] + " : " + kv[1]);
```

The above example demonstrates some of the ES6 Map methods.

ES6 Map & Set

Map supports different data types i.e. 1 and "1" are two different keys/values.

A Set object can be used to hold unique values (no repetitions are allowed).

A value in a set can be anything (objects and primitive values).

The syntax new Set([iterable]) creates a Set object where iterable is an array or any other iterable object of values.

The size property returns the number of distinct values in a set.

For example:

```
let set = new Set([1, 2, 4, 2, 59, 9, 4, 9, 1]);

console.log(set.size); // 5
```

ES6 Map & Set

Methods

- add(value) Adds a new element with the given value to the Set.
- delete(value) Deletes a specified value from the set.
- has(value) Returns true if a specified value exists in the set and false otherwise.
- clear() Clears the set.
- values() Returns an Iterator of values in the set.

ES6 Map & Set

For example:

```
let set = new Set();
set.add(5).add(9).add(59).add(9);
console.log(set.has(9));
for (let v of set.values())
    console.log(v);
```

The above example demonstrates some of the ES6 Set methods.

Set supports different data types i.e. 1 and "1" are two different values.

NaN and undefined can also be stored in Set.

Task

ES6 Map

Five employees at a company are stored in Map in the program you are given. Their names are set as keys and their positions as values. The company is hiring one more employee. The program should take the name and the position as inputs and store them in the existing map. Complete the program to perform that operation and output to the console the list of employees in the format shown in the sample output.

Sample Input

```
Bob
Developer
```

Sample Output

```
Richard : Developer
Maria : SEO Specialist
Tom : Product Manager
David : Accountant
Sophia : HR Manager
Bob : Developer
```

The **entries()** method returns an Iterator of array[key, value] in the map for each element.
Don't forget to put spaces before and after the colon (:) in the output.

More on ES6

A Promise is a better way for asynchronous programming when compared to the common way of using a setTimeout() type of method.

Consider this example:

```
setTimeout(function() {
    console.log("Work 1");
    setTimeout(function() {
        console.log("Work 2");
    }, 1000);
}, 1000);
console.log("End");
```

It prints "End", "Work 1" and "Work 2" in that order (the work is done asynchronously).

But if there are more events like this, the code becomes very complex.

More on ES6

ES6 comes to the rescue in such situations. A promise can be created as follows:

```
new Promise(function(resolve, reject) {
    // Work
    if (success)
        resolve(result);
    else
        reject(Error("failure"));
});
```

Here, resolve is the method for success and reject is the method for failure.

If a method returns a promise, its calls should use the then method which takes two methods as input; one for success and the other for the failure.

More on ES6

For Example:

```
function asyncFunc(work) {
    return new Promise(function(resolve, reject) {
        if (work === "")
            reject(Error("Nothing"));
        setTimeout(function() {
            resolve(work);
        }, 1000);
    });
}
```

It also prints "End", "Work 1" and "Work 2" (the work is done asynchronously).

But, this is clearly more readable than the previous example and in more complex situations it is easier to work with.

Run the code and see how it works!

```
asyncFunc("Work 1") // Task 1
.then(function(result) {
    console.log(result);
    return asyncFunc("Work 2"); // Task 2
}, function(error) {
    console.log(error);
})
.then(function(result) {
    console.log(result);
}, function(error) {
    console.log(error);
});
console.log("End");
```

More on ES6

Symbol.iterator is the default iterator for an object. The for...of loops are based on this type of iterator.

In the example below, we will see how we should implement it and how generator functions are used.

Example:

```
let myIterableObj = {
  [Symbol.iterator] : function* () {
    yield 1; yield 2; yield 3;
  }
};
console.log([...myIterableObj]); // [ 1, 2, 3 ]
```

First, we create an object, and use the Symbol.iterator and generator function to fill it with some values.

In the second line of the code, we use a * with the function keyword. It's called a generator function (or gen function).

More on ES6

For example, here is a simple case of how gen functions can be useful:

```
function* idMaker() {
  let index = 0;
  while (index < 5)
    yield index++;
}
var gen = idMaker();
console.log(gen.next().value);
console.log(gen.next().value);
console.log(gen.next().value);
console.log(gen.next().value);
console.log(gen.next().value);
// Try to add one more console.log, just like the above see what happens.
```

More on ES6

We can exit and re-enter generator functions later. Their variable bindings (context) will be saved across re-entrances.

They are a very powerful tool for asynchronous programming, especially when combined with Promises.

They can also be useful for creating loops with special requirements.

More on ES6

We can nest generator functions inside each other to create more complex structures and pass them arguments while we are calling them.

The example below will show a useful case of how we can use generator functions and Symbol.iterators together.

Example:

```
const arr = ['0', '1', '4', 'a', '9', 'c', '16'];

const my_obj = {

  [Symbol.iterator]: function*() {

    for(let index of arr) {

      yield `${index}`;

    }

  }

};

const all = [...my_obj] /* Here you can replace the '[...my_obj]' with 'arr'. */

  .map(i => parseInt(i, 10))

  .map(Math.sqrt)

  .filter((i) => i < 5) /* try changing the value of 5 to 4 see what happens.*/

  .reduce((i, d) => i + d); /* comment this line while you are changing the value of the line above */

console.log(all);
```

We create an object of 7 elements by using Symbol.iterator and generator functions.

In the second part, we assign our object to a constant all. At the end, we print its value.

Run the code and see how it works!

More on ES6

It is a good practice to divide your related code into modules.

Before ES6 there were some libraries which made this possible (e.g., RequireJS, CommonJS). ES6 is now supporting this feature natively.

Considerations when using modules:

The first consideration is maintainability.

A module is independent of other modules, making improvements and expansion possible without any dependency on code in other modules.

The second consideration is namespacing.

In an earlier lesson, we talked about variables and scope.

As you know, vars are globally declared, so it's common to have namespace pollution where unrelated variables are accessible all over our code. Modules solve this problem by creating a private space for variables.

Another important consideration is reusability.

When we write code that can be used in other projects,

modules make it possible to easily reuse the code without having to rewrite it in a new project.

More on ES6

Let's see how we should use modules in JS files.

For Example:

```
// lib/math.js
export let sum = (x, y) => { return x + y; }
export let pi = 3.14;
// app.js
import * as math from "lib/math"
console.log(`2p = + ${math.sum(math.pi, math.pi)}`)
```

Here we are exporting the sum function and the pi variable so we can use them in different files.

ES6 supports modules officially, however, some browsers are not supporting modules natively yet.

So, we should use bundlers (builders) such as Webpack or Browserify to run our code.

ES6 also introduced new built-in methods to make several tasks easier. Here we will cover the most common ones.

More on ES6

Array Element Finding

The legacy way to find the first element of an array by its value and a rule was the following:

```
let res = [4, 5, 1, 8, 2, 0].filter(function (x) {
  return x > 3;
})[0];
console.log(res);
```

The new syntax is cleaner and more robust:

```
let res =
[4, 5, 1, 8, 2, 0].find(x => x > 3);
console.log(res);
```

You can also get the index of the item above by using the findIndex() method:

```
let res =
[4, 5, 1, 8, 2, 0].findIndex(x => x > 3);
console.log(res);
```

More on ES6

Repeating Strings

Before ES6 the following syntax was the correct way to repeat a string multiple times:

```
console.log(Array(3 + 1).join("foo"));
```

With the new syntax, it becomes:

```
console.log("foo".repeat(3));
```

Searching Strings

Before ES6 we only used the indexOf() method to find the position of the text in the string.

It is always a good practice to refactor your code with the new syntax to learn new things and make your code more understandable.