

Welcome to JavaScript

JavaScript is one of the most popular programming languages on earth and is used to add interactivity to webpages, process data, as well as create various applications (mobile apps, desktop apps, games, and more)

Your First JavaScript

Let's start with adding JavaScript to a webpage.

JavaScript on the web lives inside the **HTML** document.

In HTML, JavaScript code must be inserted between **<script>** and **</script>** tags:

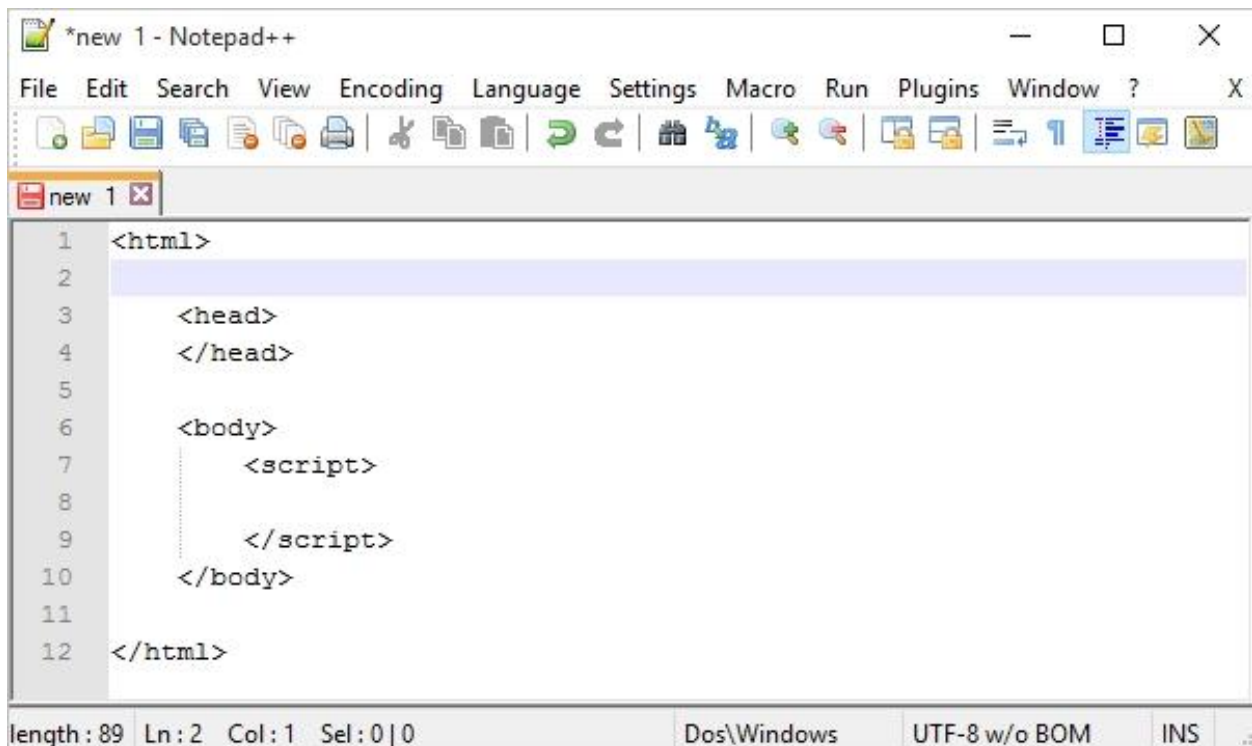
```
<script>
```

...

```
</script>
```

JavaScript can be placed in the HTML page's **<body>** and **<head>** sections.

In the example below, we placed it within the **<body>** tag.



The screenshot shows a Notepad++ window titled '*new 1 - Notepad++'. The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Macro, Run, Plugins, Window, and ?. The toolbar contains various icons for file operations and editing. The text area shows the following HTML code:

```
1 <html>
2
3   <head>
4   </head>
5
6   <body>
7       <script>
8
9       </script>
10  </body>
11
12 </html>
```

The status bar at the bottom displays: length: 89, Ln: 2, Col: 1, Sel: 0 | 0, Dos\Windows, UTF-8 w/o BOM, and INS.

Remember that the script, which is placed in the head section, will be executed before the <body> is rendered. If you want to get elements in the body, it's a good idea to place your script at the end of the body tag.

Output

Let's use JavaScript to print "Hello World" to the browser.

```
<html>
<head> </head>
<body>
<script>
document.write("Hello World!");
</script>
</body>
</html>
```

The **document.write()** function writes a string into our HTML document. This function can be used to write text, HTML, or both.

Formatting Text

Just like in HTML, we can use HTML tags to format text in JavaScript. For example, we can output the text as a heading.

```
<html>
<head> </head>
<body>
<script>
document.write("<b1>Hello World!</b1>");
</script>
</body>
</html>
```

JavaScript in <head>

You can place any number of scripts in an HTML document. Typically, the script tag is placed in the head of the HTML document:

```
<html>
<head>
<script>
</script>
</head>
<body>
</body>
</html>
```

There is also a `<noscript>` tag. Its content will be shown if the client's browser doesn't support JS scripts.

JavaScript in `<body>`

Alternatively, include the JavaScript in the `<body>` tag.

```
<html>
<head> </head>
<body>
<script>
</script>
</body>
</html>
```

The `<script>` Tag

The `<script>` tag can take two attributes, **language** and **type**, which specify the script's type:

```
<script language="javascript" type="text/javascript">

</script>
```

In the example below, we created an alert box inside the script tag, using the **alert()** function.

```
<html>
<head>
<title></title>
```

```
<script type="text/javascript">  
alert("This is an alert box!");  
</script>  
</head>  
<body>  
</body>  
</html>
```

External JavaScript

Scripts can also be placed in **external files**.

External scripts are useful and practical when the same code is used in a number of different web pages.

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

Below, we've created a new **text file**, and called it **demo.js**.

Having JS scripts in separate files makes your code much readable and clearer.

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

Here is an example:

```
<html>  
<head>  
<title> </title>  
<script src="demo.js"></script>  
</head>  
<body>  
</body>  
</html>
```

Your **demo.js** file includes the following JavaScript:

```
alert("This is an alert box!");
```

External scripts cannot contain `<script>` tags.

Placing a JavaScript in an external file has the following advantages:

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

JavaScript Comments

Not all JavaScript statements are "executed".

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

Comments are ignored, and are not executed.

Single line comments use double slashes.

```
<script>  
// This is a single line comment  
alert("This is an alert box!");  
</script>
```

It's a good idea to make a comment about the logic of large functions to make your code more readable for others.

Multiple-Line Comments

Everything you write between /*and */ will be considered as a multi-line comment.

Here is an example.

```
<script>  
/* This code  
creates an  
alert box */  
alert("This is an alert box!");  
</script>
```

Comments are used to describe and explain what the code is doing.

Variables

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

Use the **var** keyword to declare a variable:

```
var x = 10;
```

In the example above, the value **10** is assigned to the variable **x**.

JavaScript is case sensitive. For example, the variables *lastName* and *lastname*, are two different variables.

The Equal Sign

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

For example, **x = y** will assign the value of **y** to **x**.

A variable can be declared without a value. The value might require some calculation, something that will be provided later, like user input.

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

Using Variables

Let's assign a value to a variable and output it to the browser.

```
var x = 100;  
document.write(x);
```

Using variables is useful in many ways. You might have a thousand lines of code that may include the variable **x**. When you change the value of **x** **one time**, it will automatically be changed in **all places** where you used it.

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

Naming Variables

JavaScript variable names are case-sensitive.

In the example below we changed x to uppercase:

```
var x = 100;  
document.write(X);
```

This code will not result in any output, as x and X are two different variables.

Naming rules:

- The first character **must be** a letter, an underscore (_), or a dollar sign (\$). Subsequent characters may be letters, digits, underscores, or dollar signs.
- Numbers are **not allowed** as the first character.
- Variable names **cannot** include a **mathematical or logical operator** in the name. For instance, *2*something* or *this+that*;
- JavaScript names **must not contain spaces**.

Hyphens are not allowed in JavaScript. It is reserved for subtractions.

Naming Variables

There are some other rules to follow when naming your JavaScript variables:

- You **must not** use any **special symbols**, like *my#num*, *num%*, etc.
- Be sure that you do not use any of the following JavaScript reserved words.

Reserved Words in JavaScript

| | | | |
|----------|------------|------------|--------------|
| abstract | else | instanceof | switch |
| boolean | enum | int | synchronized |
| break | export | interface | this |
| byte | extends | long | throw |
| case | false | native | throws |
| catch | final | new | transient |
| char | finally | null | true |
| class | float | package | try |
| const | for | private | typeof |
| continue | function | protected | var |
| debugger | goto | public | void |
| default | if | return | volatile |
| delete | implements | short | while |
| do | import | static | with |
| double | in | super | |

When you get more familiar with JavaScript, remembering these keywords will be much easier.

Data Types

The term **data type** refers to the types of values with which a program can work. JavaScript variables can hold many data types, such as **numbers**, **strings**, **arrays**, and more.

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

JavaScript numbers can be written with or without decimals.

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

This variable can be easily changed to other types by assigning to it any other data type value, like `num = 'some random string'`.

Floating-Point Numbers

JavaScript numbers can also have decimals.

```
<script>
var price = 55.55;
document.write(price);
</script>
```

JavaScript numbers are always stored as **double precision floating point numbers**.

Strings

JavaScript **strings** are used for storing and manipulating text.

A string can be any text that appears within **quotes**. You can use single or double quotes.

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

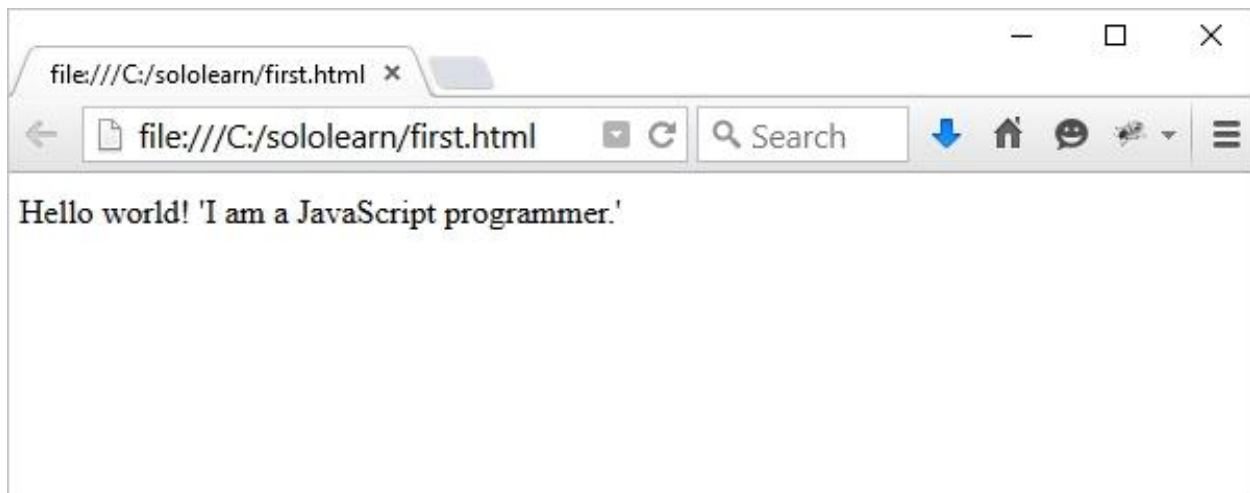
You can use quotes inside a string, as long as they don't match the quotes surrounding the string.

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

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

As strings must be written within quotes, quotes inside the string must be handled. The **backslash (\) escape character** turns special characters into string characters.

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



The escape character (\) can also be used to insert other special characters into a string.

These special characters can be added to a text string using the backslash sign.

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

If you begin a string with a single quote, then you should also end it with a single quote. The same rule applies to double quotes. Otherwise, JavaScript will become confused.

Booleans

In JavaScript Boolean, you can have one of two values, either **true** or **false**. These are useful when you need a data type that can only have one of two values, such as Yes/No, On/Off, True/False.

Example:

```
var isActive = true;  
var isHoliday = false;
```

The Boolean value of 0 (zero), null, undefined, empty string is **false**. Everything with a "real" value is **true**.

Arithmetic Operators

Arithmetic operators perform arithmetic functions on numbers (literals or 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 |

In the example below, the addition operator is used to determine the sum of two numbers.

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

// Outputs 15

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);
```

//Outputs 6630

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.

Multiplication

The multiplication operator (*) multiplies one number by the other.

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

// Outputs 50

10 * '5' or '10' * '5' gives the same result. Multiplying a number with string values like 'sololearn' * 5 returns NaN (Not a Number).

Division

The / operator is used to perform division operations:

```
var x = 100 / 5;  
document.write(x);
```

// Outputs 20

Remember to handle cases where there could be a division by 0.

The Modulus

Modulus (%) operator returns the division remainder (what is left over).

```
var myVariable = 26 % 6;
```

```
//myVariable equals 2
```

In JavaScript, the modulus operator is used not only on integers, but also on floating point numbers.

Increment & Decrement

Increment ++

The increment operator increments the numeric value of its operand by one. If placed before the operand, it returns the incremented value. If placed after the operand, it returns the original value and then increments the operand.

Decrement --

The decrement operator decrements the numeric value of its operand by one. If placed before the operand, it returns the decremented value. If placed after the operand, it returns the original value and then decrements the operand.

| 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 |

As in school mathematics, you can change the order of the arithmetic operations by using parentheses.

Example: `var x = (100 + 50) * 3;`

Assignment Operators

Assignment operators assign values to JavaScript variables.

| Operator | Example | Is equivalent to |
|-----------------|---------------------|------------------------|
| <code>=</code> | <code>x = y</code> | <code>x = y</code> |
| <code>+=</code> | <code>x += y</code> | <code>x = x + y</code> |
| <code>-=</code> | <code>x -= y</code> | <code>x = x - y</code> |
| <code>*=</code> | <code>x *= y</code> | <code>x = x * y</code> |
| <code>/=</code> | <code>x /= y</code> | <code>x = x / y</code> |
| <code>%=</code> | <code>x %= y</code> | <code>x = x % y</code> |

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

Comparison Operators

Comparison operators are used in logical statements to determine equality or difference between variables or values. They return **true** or **false**.

The **equal to** (`==`) operator checks whether the operands' values are equal.

```
var num = 10;
```

```
// num == 8 will return false
```

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

The table below explains the comparison operators.

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

When using operators, be sure that the arguments are of the same data type; numbers should be compared with numbers, strings with strings, and so on.

Logical Operators

Logical Operators, also known as **Boolean Operators**, evaluate the expression and return **true** or **false**.

The table below explains the logical operators (**AND**, **OR**, **NOT**).

| Logical Operators | |
|-------------------------|--|
| <code>&&</code> | Returns true, if both operands are true |
| <code> </code> | Returns true, if one of the operands is true |
| <code>!</code> | Returns true, if the operand is false, and false, if the operand is true |

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

In the following example, we have connected two Boolean expressions with the **AND** operator.

```
(4 > 2) && (10 < 15)
```

For this expression to be **true**, both conditions must 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**.
- Based on these results, the whole expression is found to be **true**.

Conditional (Ternary) Operator

Another JavaScript conditional operator assigns a value to a variable, based on some condition.

Syntax: variable = (condition) ? value1: value2

For example:

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

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".

Logical operators allow you to connect as many expressions as you wish.

String Operators

The most useful operator for strings is *concatenation*, represented by the + sign. Concatenation can be used to build strings by joining together multiple strings, or by joining strings with other types:

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

The above example declares and initializes two string variables, and then concatenates them.

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

Conditionals and Loops

The if Statement

Very often when you write code, you want to perform different actions based on different conditions.

You can do this by using **conditional statements** in your code.

Use **if** to specify a block of code that will be executed if a specified condition is true.

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

The statements will be executed only if the specified condition is **true**.

Example:

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

This is another example of a **false** conditional statement.

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

As the condition evaluates to false, the alert statement is skipped and the program continues with the line after the if statement's closing curly brace.

Note that **if** is in lowercase letters. Uppercase letters (If or IF) will generate an error.

The else Statement

Use the **else** statement to specify a block of code that will execute if the condition is **false**.

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

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

The example below demonstrates the use of an **if...else** statement.

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

the above example says:

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

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

There is also another way to do this check using the ? operator: `a > b ? alert(a) : alert(b)`.

else if

You can use the **else if statement** to specify a new condition if the first condition is false.

Example:

```
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 above code says:

- **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 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.

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.

Switch

In cases when you need to test for multiple conditions, writing **if else** statements for each condition might not be the best solution.

The **switch statement** is used to perform different actions based on different conditions.

Syntax:

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

The switch expression is evaluated once. The value of the expression is compared with the values of each **case**. If there is a match, the associated block of code is executed.

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

Consider the following 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");  
}  
  
// Outputs "Tuesday"
```

You can have as many **case** statements as needed.

The break Keyword

When JavaScript code reaches a **break** keyword, it breaks out of the switch block. This will stop the execution of more code and case testing inside the block.

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

The default Keyword

The **default** keyword specifies the code to run if there is no case match.

```
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.");
}
```

//Outputs "Color not found."

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

Loops

Loops can execute a block of code a number of times. They are handy in cases in which 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**.

The **for** loop is commonly used when creating a loop.

The syntax:

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

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.

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

The For Loop

The example below creates a **for** loop that prints numbers 1 through 5.

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

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 (i 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.

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

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

Also, you can 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 />");
}
```

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

Statement 2 is also optional.

If you omit statement 2, you must provide 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), or anything else.

Statement 3 is also optional, and it can be omitted if you increment your values inside the loop.

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

You can have multiple nested for loops.

The While Loop

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

Syntax:

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

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

Consider the following example.

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

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

This will output the values from 0 to 10.

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

If you forget to increase the variable used in the condition, the loop will never end.

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

The Do...While Loop

The **do...while** loop is a variant of the while loop. 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.

Syntax:

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


Note the **semicolon** used at the end of the do...while loop.

Example:

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

This prints out numbers from 20 to 25.

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.

Break and Continue

Break

The **break** statement "jumps out" of a loop and continues executing the code after the loop.

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

Once i reaches 5, it will break out of the loop.

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

Continue

The **continue** statement breaks only one iteration in the loop, and continues with the next iteration.

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

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

Functions

User-Defined Functions

JavaScript 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.

Defining a Function

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
```

```
}
```

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

Calling a Function

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();  
//Alerts "Calling a Function!"
```

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

Calling 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();  
// "Alert box!"
```

```
// some other code
```

```
myFunction();  
// "Alert box!"
```

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.

Using Parameters

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

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

```
sayHello("David");  
//Alerts "Hi, 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.

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.

Multiple Parameters

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.

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)  
//Outputs "John is 20 years old."
```

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.

Function Return

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 myFunction(a, b) {  
  return a * b;  
}
```

```
var x = myFunction(5, 6);  
// Return value will end up in x  
// x equals 30
```

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

Another example:

```
function addNumbers(a, b) {  
  var c = a+b;  
  return c;  
}  
document.write( addNumbers(40, 2) );  
//Outputs 42
```

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

Alert, Prompt, Confirm

The Alert Box

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.

Example:

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

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

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

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

Prompt Box

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).

Example:

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

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.

Confirm Box

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**.

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");  
}
```

Introducing Objects

JavaScript 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  
};
```

These values are called **properties**.

| Property | Property Value |
|----------|----------------|
| name | John |
| age | 31 |
| favColor | green |
| height | 183 |

JavaScript objects are containers for **named values**.

Object Properties

You can access object properties in two ways.

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

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'];
```

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);  
//Outputs 2
```

Objects are one of the core concepts in JavaScript.

Object Methods

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.

The Object Constructor

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 Objects

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

```
var p1 = new person("John", 42, "green");  
var p2 = new person("Amy", 21, "red");
```

```
document.write(p1.age); // Outputs 42  
document.write(p2.name); // Outputs "Amy"
```

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

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);
```

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

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

Understanding the creation of objects is essential.

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.

Using Object Initializers

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.

```
document.write(John.age);  
//Outputs 25
```

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

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.

For Example:

```
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");  
//Now p.name equals to "John"
```

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.

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.

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());  
// Outputs 1994
```

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

JavaScript 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.

Accessing an Array

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]);
```

```
//Outputs "undefined"
```

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

Creating 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";
```

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.

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++";
```

You can add as many elements as you need to.

Array Literal

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

```
var courses = ["HTML", "CSS", "JS"];
```

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.

The length Property

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);  
//Outputs 3
```

The **length** property is always one more than the highest array index. If the array is empty, the length property returns **0**.

Combining Arrays

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);
```

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.

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"]);
//Outputs "46"
```

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

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

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 |

For example:

```
document.write(Math.PI);  
//Outputs 3.141592653589793
```

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 |

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

```
var number = Math.sqrt(4);
document.write(number);
//Outputs 2
```

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.

The Date Object

`setInterval`

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

The Date Object

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 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.

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.

Date Methods

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

| Method | Description |
|--------------------------------|---------------------------|
| <code>getFullYear()</code> | gets the year |
| <code>getMonth()</code> | gets the month |
| <code>getDate()</code> | gets the day of the month |
| <code>getDay()</code> | gets the day of the week |
| <code>getHours()</code> | gets the hour |
| <code>getMinutes()</code> | gets the minutes |
| <code>getSeconds()</code> | gets the seconds |
| <code>getMilliseconds()</code> | gets the milliseconds |

For example:

```
var d = new Date();  
var hours = d.getHours();  
//hours is equal to the current hour
```

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);
```

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.

JavaScript - The Boolean Object

The **Boolean** object represents two values, either "true" or "false". If *value* parameter is omitted or is 0, -0, null, false, **NaN**, undefined, or the empty string (""), the object has an initial value of false.

Syntax

Use the following syntax to create a **boolean** object.

```
var val = new Boolean(value);
```

Boolean Properties

Here is a list of the properties of Boolean object –

1. Returns a reference to the Boolean function that created the object.
2. The prototype property allows you to add properties and methods to an object.

In the following sections, we will have a few examples to illustrate the properties of Boolean object.

Boolean Methods

Here is a list of the methods of Boolean object and their description.

JavaScript boolean **toSource()** method returns a string representing the source code of the object.

Note – This method is not compatible with all the browsers.

Syntax

Its syntax is as follows –

```
boolean.toSource()
```

Return Value

Returns a string representing the source code of the object.

Example

Try the following example.

```
<html>
  <head>
    <title>JavaScript toSource() Method</title>
  </head>
  <body>
    <script type = "text/javascript">
      function book(title, publisher, price) {
        this.title = title;
        this.publisher = publisher;
        this.price = price;
      }
      var newBook = new book("Perl","Leo Inc",200);
      document.write(newBook.toSource());
    </script>
  </body>
</html>
```

Output

```
({title:"Perl", publisher:"Leo Inc", price:200})
```

Description

toString():

This method returns a string of either "true" or "false" depending upon the value of the object.

Syntax

Its syntax is as follows –

```
boolean.toString()
```

Return Value

Returns a string representing the specified Boolean object.

Example

Try the following example.

```
<html>
  <head>
    <title>JavaScript toString() Method</title>
  </head>

  <body>
    <script type = "text/javascript">
      var flag = new Boolean(false);
      document.write( "flag.toString is : " + flag.toString() );
    </script>
  </body>
</html>
```

Output

flag.toString is : false

Description

JavaScript boolean **valueOf()** method returns the primitive value of the specified **boolean** object.

Syntax

Its syntax is as follows –

boolean.valueOf()

Return Value

Returns the primitive value of the specified **boolean** object.

Example

Try the following example.

```
<html>
  <head>
    <title>JavaScript valueOf() Method</title>
  </head>

  <body>
    <script type = "text/javascript">
      var flag = new Boolean(false);
      document.write( "flag.valueOf is : " + flag.valueOf() );
    </script>
  </body>
</html>
```

Output

flag.valueOf is : false

JavaScript Errors Handling:

Definition and Usage

The try/catch/finally statement handles some or all of the errors that may occur in a block of code, while still running code.

Errors can be coding errors made by the programmer, errors due to wrong input, and other unforeseeable things.

The **try** statement allows you to define a block of code to be tested for errors while it is being executed.

The **catch** statement allows you to define a block of code to be executed, if an error occurs in the **try** block.

The **finally** statement lets you execute code, after try and catch, regardless of the result.

Note: The catch and finally statements are both optional, but you need to use one of them (if not both) while using the try statement.

Tip: When an error occurs, JavaScript will normally stop, and generate an error message. Use the [throw](#) statement to create a custom error (throw an exception). If you use **throw** together with **try** and **catch**, you can control program flow and generate custom error messages.

Syntax

```
try {  
    tryCode - Block of code to try  
}  
catch(err) {  
    catchCode - Block of code to handle errors  
}  
finally {  
    finallyCode - Block of code to be executed regardless of the try / catch result  
}
```

Errors Will Happen!

When executing JavaScript code, different errors can occur.

Errors can be coding errors made by the programmer, errors due to wrong input, and other unforeseeable things.

Example

In this example we have written alert as adddlert to deliberately produce an error:

```
<script>  
try {  
    adddlert("Welcome guest!");  
}  
catch(err) {  
    document.getElementById("demo").innerHTML = err.message;
```

```
}  
</script>
```

JavaScript catches **addlert** as an error, and executes the catch code to handle it.
JavaScript try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The JavaScript statements try and catch come in pairs:

```
try {  
    Block of code to try  
}  
catch(err) {  
    Block of code to handle errors  
}
```

JavaScript Throws Errors

When an error occurs, JavaScript will normally stop and generate an error message.

The technical term for this is: JavaScript will **throw an exception (throw an error)**.

JavaScript will actually create an **Error object** with two properties: **name** and **message**.

The throw Statement

The throw statement allows you to create a custom error.

Technically you can **throw an exception (throw an error)**.

The exception can be a JavaScript String, a Number, a Boolean or an Object:

```
throw "Too big"; // throw a text  
throw 500;       // throw a number
```

If you use throw together with try and catch, you can control program flow and generate custom error messages.

```
try {  
  
    alert('Start of try runs'); // (1) <--  
  
    // ...no errors here  
  
    alert('End of try runs'); // (2) <--  
  
} catch(err) {  
  
    alert('Catch is ignored, because there are no errors'); // (3)  
  
}
```

Next example:

```
try {  
  
    alert('Start of try runs'); // (1) <--  
  
    lalala; // error, variable is not defined!  
  
    alert('End of try (never reached)'); // (2)  
  
} catch(err) {  
  
    alert(`Error has occurred!`); // (3) <--  
  
}
```

Throw example:

```
<!DOCTYPE html>  
<html>
```

```
<body>

<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="message"></p>

<script>
function myFunction() {
  var message, x;
  message = document.getElementById("message");
  message.innerHTML = "";
  x = document.getElementById("demo").value;
  try {
    if(x == "") throw "is Empty";
    if(isNaN(x)) throw "not a number";
    if(x > 10) throw "too high";
    if(x < 5) throw "too low";
  }
  catch(err) {
    message.innerHTML = "Input " + err;
  }
}
</script>

</body>
</html>
```

Example: Date of Birth

```
<html>

<body>

<h1> Enter your Date of Birth </h1>

<table>
```

```
<tr>
<th>
<font size=3>Date of Birth: </font>
<th>
<Select name=Date>
  <option>DD
<script> for(i=1; i<=31;i++) document.write("<option>" + i); </script>
<th>
<Select name=Month>
<option>MM
<script> for(i=1; i<=12;i++) document.write("<option>" + i);</script>
<th>
<Select name=Year>
<option>YYYY<script> for(i=1901; i<=2020;i++) document.write("<option>"
+ i); </script>
</table>
</body>
</html>
```