

JAVASCRIPT

An Introduction to JavaScript

→ What is JavaScript?

The programs in this language are called scripts. They can be written right in a web page's HTML and run automatically as the page loads.

Scripts are provided and executed as plain text. They don't need special preparation or compilation to run.

In this aspect, JavaScript is very different from another language called Java.

→ Why is it called JavaScript?

When JavaScript was created, it initially had another name: "LiveScript". But Java was very popular at that time, so it was decided that positioning a new language as a "younger brother" of Java would help.

But as it evolved, JavaScript became a fully independent language with its own specification called ECMAScript, and now it has no relation to Java at all.

→ What can in-browser JavaScript do?

★ What makes JavaScript unique?

Full integration with HTML/CSS.

Simple things are done simply.

Supported by all major browsers and enabled by default.

→ Developer console

Code is prone to errors. You will quite likely make errors...
Oh, what am I talking about? You are absolutely going to make errors, at least if you're a human, not a robot.

But in the browser, users don't see errors by default. So, if something goes wrong in the script, we won't see what's broken and can't fix it.

To see errors and get a lot of other useful information about scripts, "developer tools" have been embedded in browsers.

→ Google Chrome

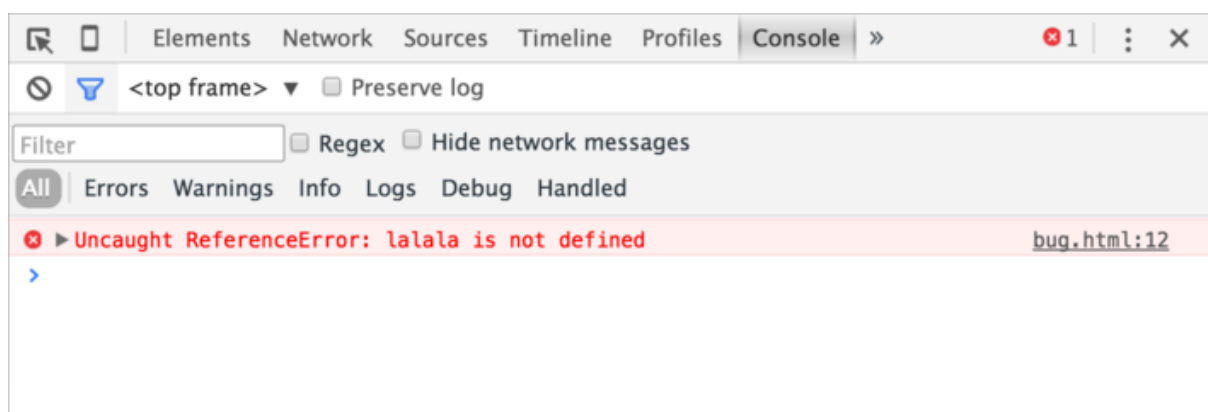
Open the page `bug.html`.

There's an error in the JavaScript code on it. It's hidden from a regular visitor's eyes, so let's open developer tools to see it.

Press F12 or, if you're on Mac, then Cmd+Opt+J.


The developer tools will open on the Console tab by default.

It looks somewhat like this:



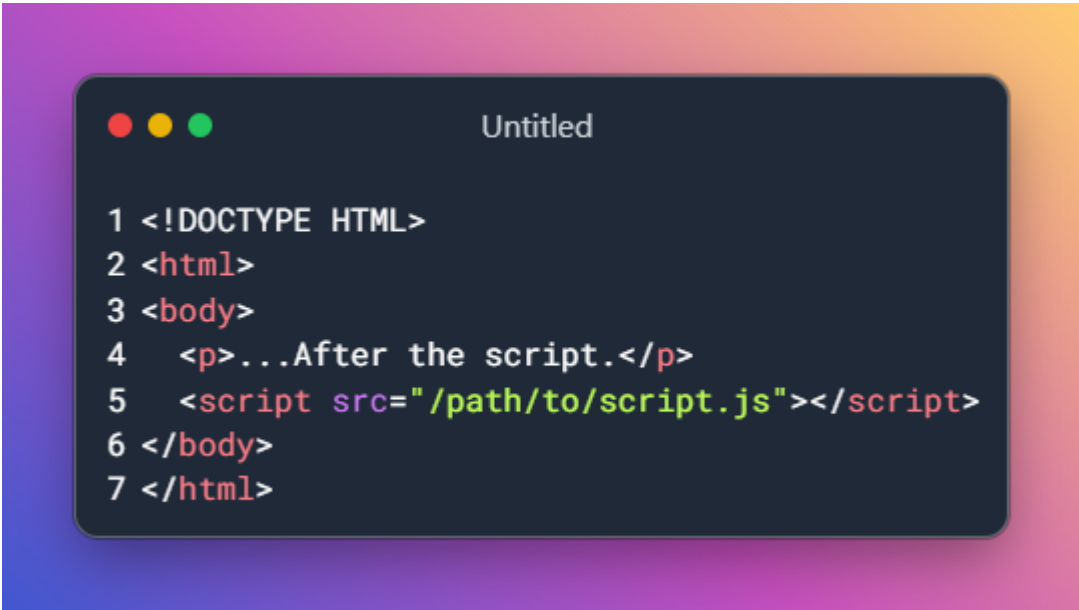
→ JavaScript Fundamentals

→ The “script” tag



```
1 <!DOCTYPE HTML>
2 <html>
3 <body>
4   <p>Before the script...</p>
5   <script>
6     alert( 'Hello, world!' );
7   </script>
8   <p>...After the script.</p>
9 </body>
10 </html>
```

→ External scripts



```
1 <!DOCTYPE HTML>
2 <html>
3 <body>
4   <p>...After the script.</p>
5   <script src="/path/to/script.js"></script>
6 </body>
7 </html>
```

→ Code structure

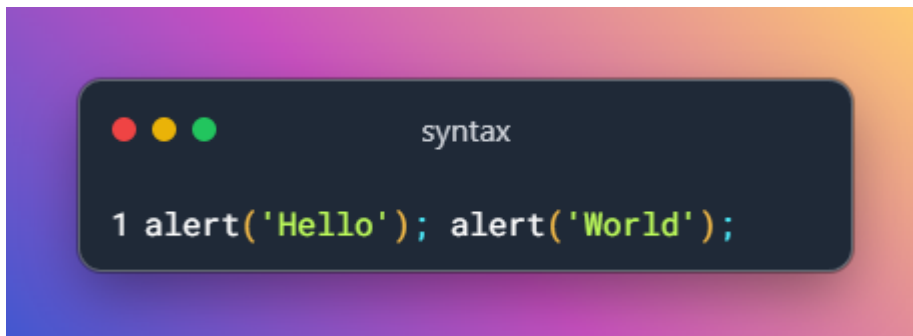
→ Statements

Statements are syntax constructs and commands that perform actions.

We've already seen a statement, `alert('Hello, world!')`, which shows the message "Hello, world!".

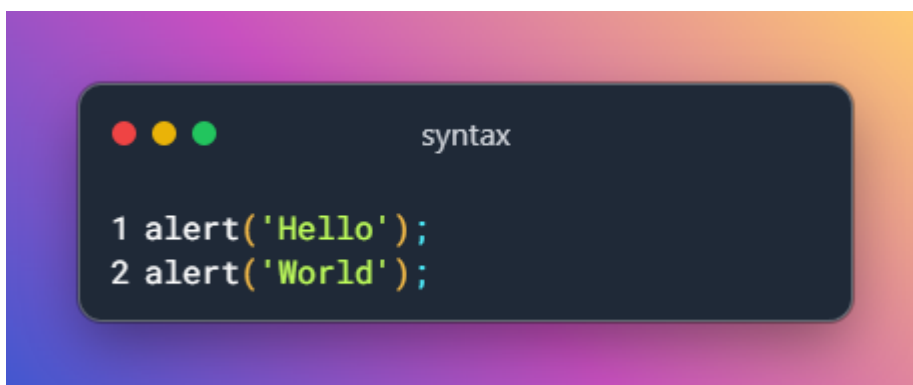
We can have as many statements in our code as we want. Statements can be separated with a semicolon.

For example, here we split "Hello World" into two alerts:



```
1 alert('Hello'); alert('World');
```

Usually, statements are written on separate lines to make the code more readable:

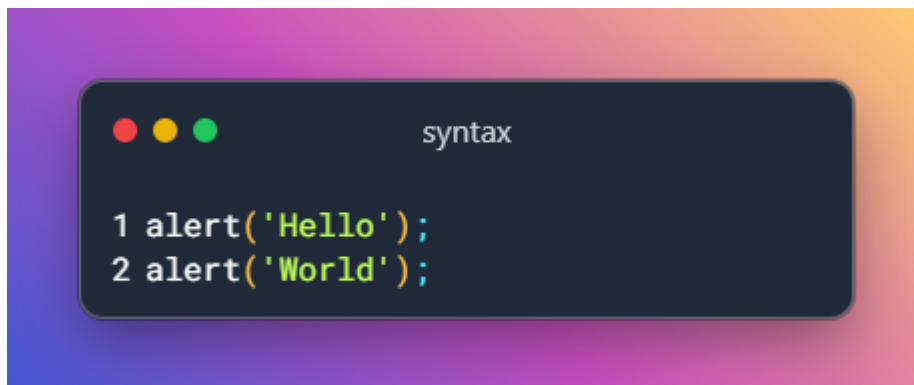


```
1 alert('Hello');  
2 alert('World');
```

→ Semicolons

A semicolon may be omitted in most cases when a line break exists.

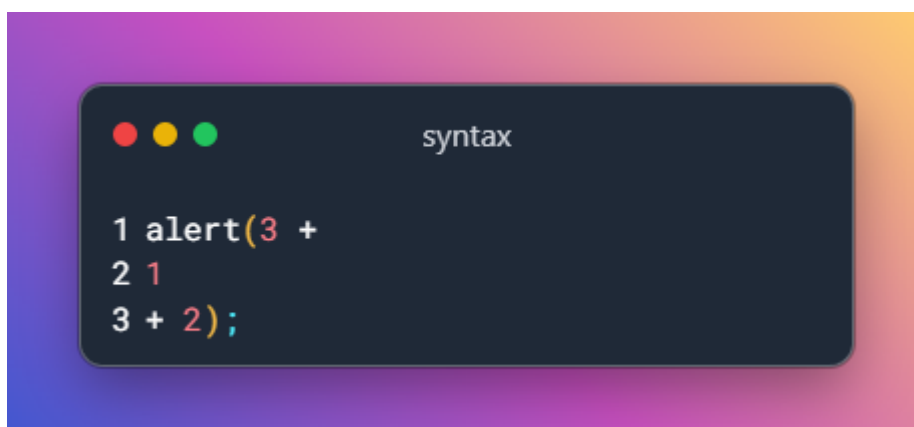
This would also work:



Here, JavaScript interprets the line break as an “implicit” semicolon. This is called an automatic semicolon insertion.

In most cases, a newline implies a semicolon. But “in most cases” does not mean “always”!

There are cases when a newline does not mean a semicolon. For example:

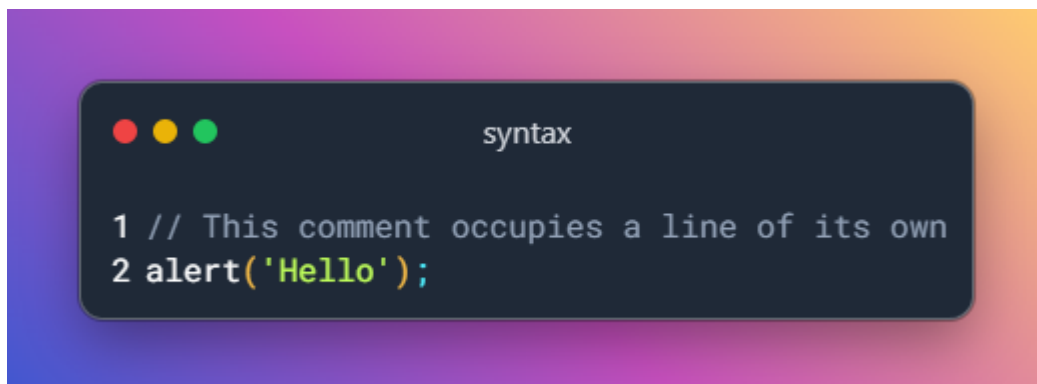


→ Comments

One-line comments start with two forward slash characters `//`.

The rest of the line is a comment. It may occupy a full line of its own or follow a statement.

Like here:

A code editor window with a dark blue background and a light blue title bar. The title bar has three colored circles (red, yellow, green) on the left and the word 'syntax' on the right. The editor contains two lines of code: line 1 is a single-line comment `// This comment occupies a line of its own` and line 2 is `alert('Hello');`.

```
1 // This comment occupies a line of its own
2 alert('Hello');
```

`alert('World');` `// This comment follows the statement`

Multiline comments start with a forward slash and an asterisk `/*` and end with an asterisk and a forward slash `*/`.

Like this:

A code editor window with a dark blue background and a light blue title bar. The title bar has three colored circles (red, yellow, green) on the left and the word 'syntax' on the right. The editor contains five lines of code: line 1 is a multiline comment `/* An example with two messages.`, line 2 is `This is a multiline comment.`, line 3 is `*/`, line 4 is `alert('Hello');`, and line 5 is `alert('World');`.

```
1 /* An example with two messages.
2 This is a multiline comment.
3 */
4 alert('Hello');
5 alert('World');
```

The content of comments is ignored, so if we put code inside `/* ... */`, it won't execute.

Sometimes it can be handy to temporarily disable a part of code:



The modern mode, "use strict"

For a long time, JavaScript evolved without compatibility issues. New features were added to the language while old functionality didn't change.

That had the benefit of never breaking existing code. But the downside was that any mistake or an imperfect decision made by JavaScript's creators got stuck in the language forever.

This was the case until 2009 when ECMAScript 5 (ES5) appeared. It added new features to the language and modified some of the existing ones. To keep the old code working, most such modifications are off by default. You need to explicitly enable them with a special directive: "use strict".

"use strict"

The directive looks like a string: "use strict" or 'use strict'. When it is located at the top of a script, the whole script works the "modern" way.

For example:

A code editor window titled "syntax" with a dark background and three colored window control buttons (red, yellow, green) in the top left. The code is as follows:

```
1 use strict;  
2  
3 // this code works the modern way
```

Ensure that "use strict" is at the top

Please make sure that "use strict" is at the top of your scripts, otherwise strict mode may not be enabled.

Strict mode isn't enabled here:

A code editor window titled "syntax" with a dark background and three colored window control buttons (red, yellow, green) in the top left. The code is as follows:

```
1  
2 alert("some code");  
3 // "use strict" below is ignored--it must be at the top  
4  
5 use strict;  
6  
7 // strict mode is not activated
```

Only comments may appear above "use strict".

There's no way to cancel use strict

There is no directive like "no use strict" that reverts the engine to old behavior.

Once we enter strict mode, there's no going back.

→ Browser console

When you use a developer console to run code, please note that it doesn't use strict by default.

Sometimes, when use strict makes a difference, you'll get incorrect results.

So, how to actually use strict in the console?

First, you can try to press Shift+Enter to input multiple lines, and put use strict on top, like this:

```
'use strict'; <Shift+Enter for a newline>  
// ...your code  
<Enter to run>
```

It works in most browsers, namely Firefox and Chrome.

If it doesn't, e.g. in an old browser, there's an ugly, but reliable way to ensure use strict. Put it inside this kind of wrapper:



→ Variables

We can declare variables to store data by using the `var`, `let`, or `const` keywords.

`let` – is a modern variable declaration.

`var` – is an old-school variable declaration. Normally we don't use it at all, but we'll cover subtle differences from `let` in the chapter The old "var", just in case you need them.

`const` – is like `let`, but the value of the variable can't be changed.

Variables should be named in a way that allows us to easily understand what's inside them.

Most of the time, a JavaScript application needs to work with information. Here are two examples:

An online shop – the information might include goods being sold and a shopping cart.

A chat application – the information might include users, messages, and much more.

Variables are used to store this information.

A variable

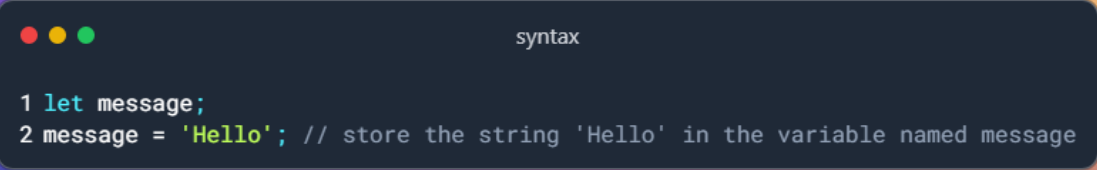
A variable is a "named storage" for data. We can use variables to store goodies, visitors, and other data.

To create a variable in JavaScript, use the `let` keyword.

The statement below creates (in other words: declares) a variable with the name "message":

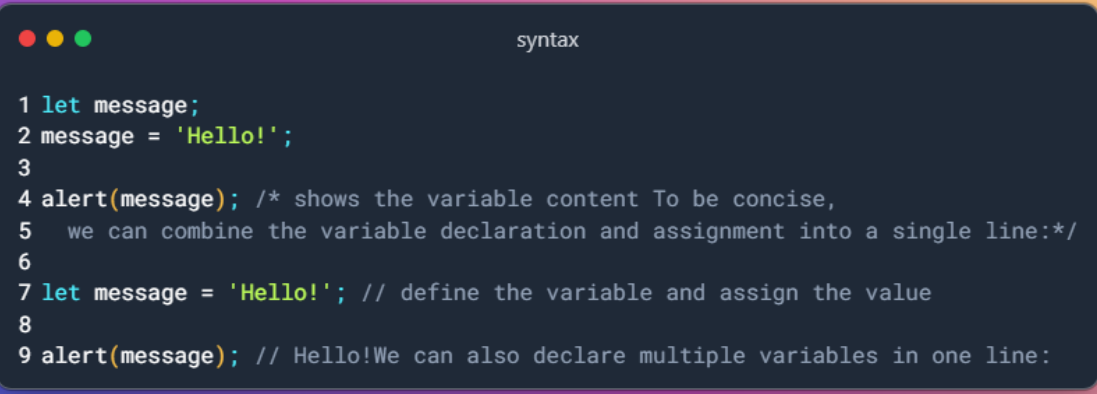
```
let message;
```

Now, we can put some data into it by using the assignment operator `=`:



```
1 let message;  
2 message = 'Hello'; // store the string 'Hello' in the variable named message
```

The string is now saved into the memory area associated with the variable. We can access it using the variable name:

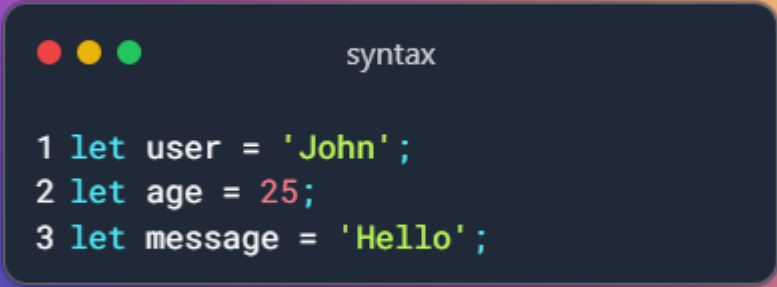


```
1 let message;  
2 message = 'Hello!';  
3  
4 alert(message); /* shows the variable content To be concise,  
5 we can combine the variable declaration and assignment into a single line:*/  
6  
7 let message = 'Hello!'; // define the variable and assign the value  
8  
9 alert(message); // Hello!We can also declare multiple variables in one line:
```

```
let user = 'John', age = 25, message = 'Hello';
```

That might seem shorter, but we don't recommend it. For the sake of better readability, please use a single line per variable.

The multiline variant is a bit longer, but easier to read:



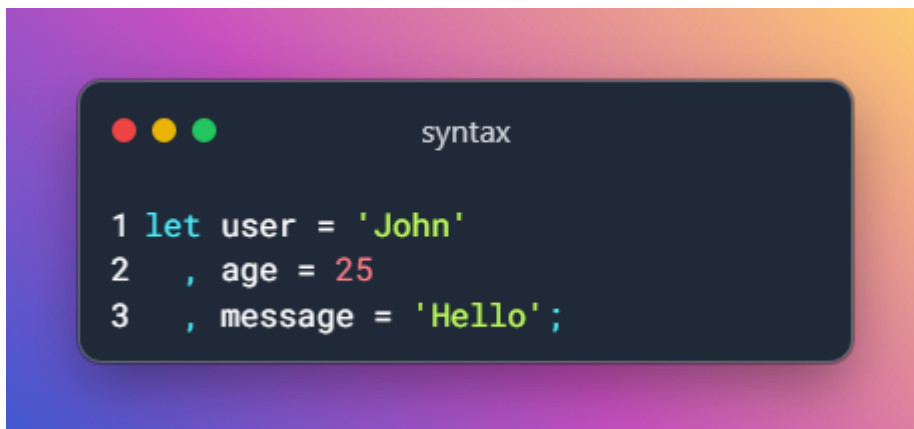
```
1 let user = 'John';  
2 let age = 25;  
3 let message = 'Hello';
```

Some people also define multiple variables in this multiline style:

A code editor window with a dark background and a light blue title bar. The title bar contains three colored circles (red, yellow, green) and the word "syntax". The editor shows three lines of JavaScript code: 1 let user = 'John', 2 age = 25, 3 message = 'Hello';. The code is color-coded: 'let' is blue, 'user' is green, 'John' is red, 'age' is green, '25' is red, 'message' is green, and 'Hello' is red. The lines are numbered 1, 2, and 3 on the left.

```
1 let user = 'John',  
2   age = 25,  
3   message = 'Hello';
```

...Or even in the “comma-first” style:

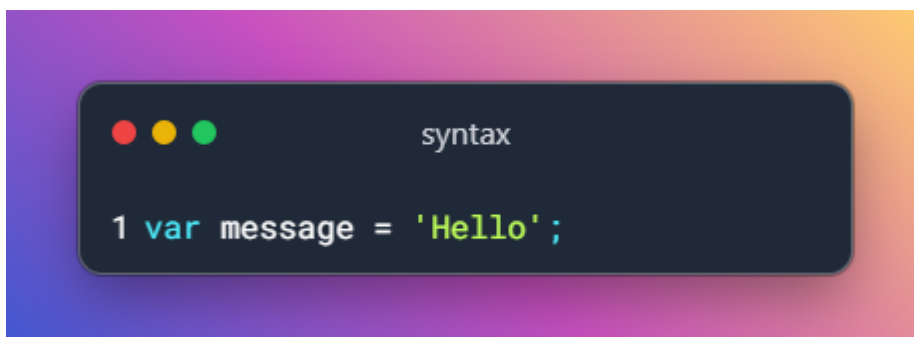
A code editor window with a dark background and a light blue title bar. The title bar contains three colored circles (red, yellow, green) and the word "syntax". The editor shows three lines of JavaScript code: 1 let user = 'John', 2 , age = 25, 3 , message = 'Hello';. The code is color-coded: 'let' is blue, 'user' is green, 'John' is red, 'age' is green, '25' is red, 'message' is green, and 'Hello' is red. The lines are numbered 1, 2, and 3 on the left.

```
1 let user = 'John'  
2   , age = 25  
3   , message = 'Hello';
```

Technically, all these variants do the same thing. So, it’s a matter of personal taste and aesthetics.

var instead of let

In older scripts, you may also find another keyword: var instead of let:

A code editor window with a dark background and a light blue title bar. The title bar contains three colored circles (red, yellow, green) and the word "syntax". The editor shows one line of JavaScript code: 1 var message = 'Hello';. The code is color-coded: 'var' is blue, 'message' is green, and 'Hello' is red. The line is numbered 1 on the left.

```
1 var message = 'Hello';
```

The var keyword is almost the same as let. It also declares a variable, but in a slightly different, “old-school” way.

There are subtle differences between `let` and `var`, but they do not matter for us yet. We'll cover them in detail in the chapter `The old "var"`.

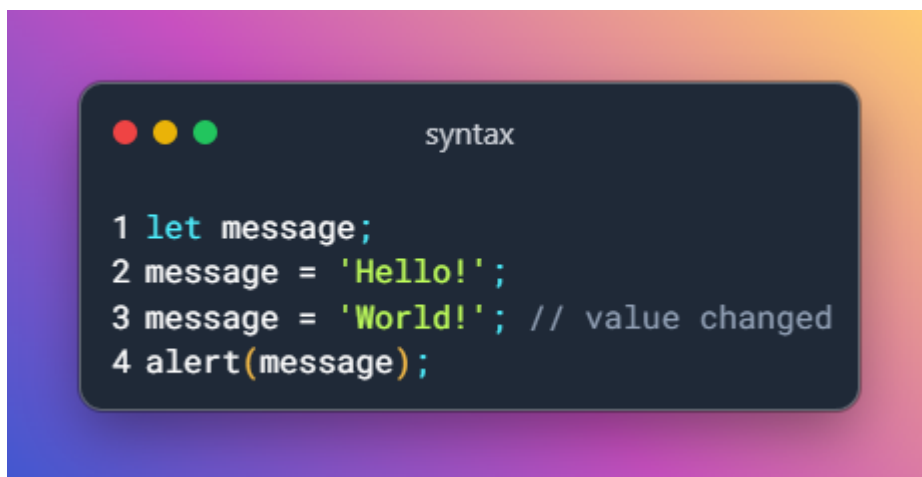
→ A real-life analogy

We can easily grasp the concept of a "variable" if we imagine it as a "box" for data, with a uniquely-named sticker on it.

For instance, the variable `message` can be imagined as a box labeled "message" with the value "Hello!" in it:

We can put any value in the box.

We can also change it as many times as we want:



When the value is changed, the old data is removed from the variable:

We can also declare two variables and copy data from one into the other.

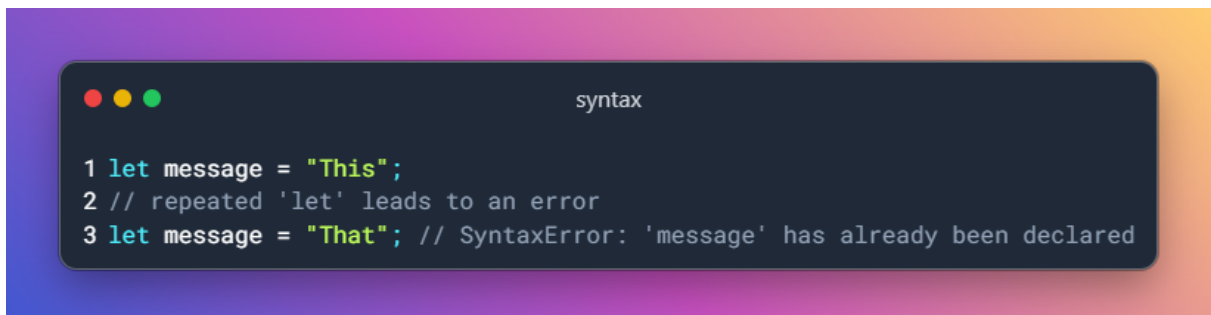


```
1 let hello = 'Hello world!';
2 let message;
3 // copy 'Hello world' from hello into message
4 message = hello;
5 // now two variables hold the same data
6 alert(hello); // Hello world!
7 alert(message); // Hello world!
```

Declaring twice triggers an error

A variable should be declared only once.

A repeated declaration of the same variable is an error:



```
1 let message = "This";
2 // repeated 'let' leads to an error
3 let message = "That"; // SyntaxError: 'message' has already been declared
```

So, we should declare a variable once and then refer to it without let.

Variable naming

There are two limitations on variable names in JavaScript:

The name must contain only letters, digits, or the symbols \$ and _.

The first character must not be a digit.

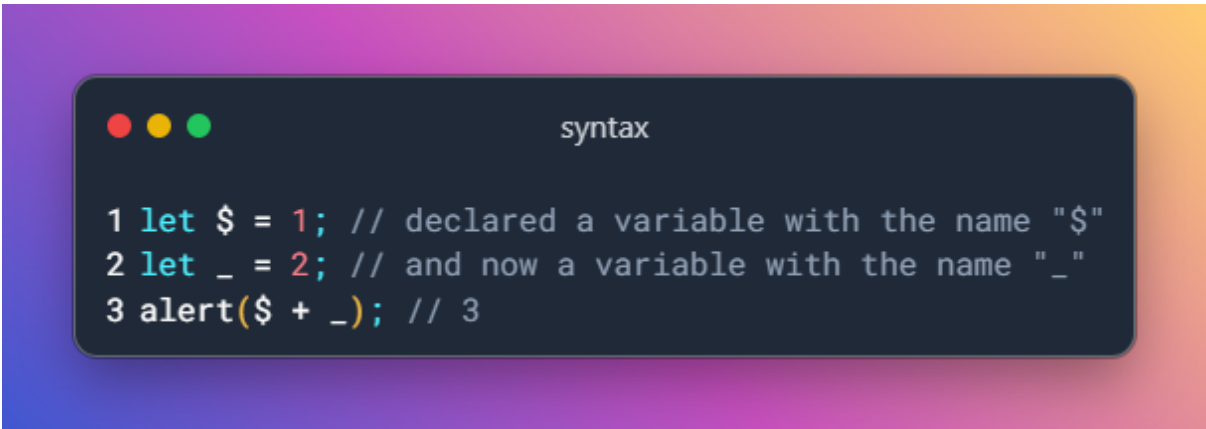
Examples of valid names:

```
let userName;  
let test123;
```

When the name contains multiple words, camelCase is commonly used. That is: words go one after another, each word except first starting with a capital letter: myVeryLongName.


What's interesting – the dollar sign '\$' and the underscore '_' can also be used in names. They are regular symbols, just like letters, without any special meaning.

These names are valid:



```
1 let $ = 1; // declared a variable with the name "$"  
2 let _ = 2; // and now a variable with the name "_"  
3 alert($ + _); // 3
```

Examples of incorrect variable names:



```
1 let 1a; // cannot start with a digit  
2 let my-name; // hyphens '-' aren't allowed in the name
```

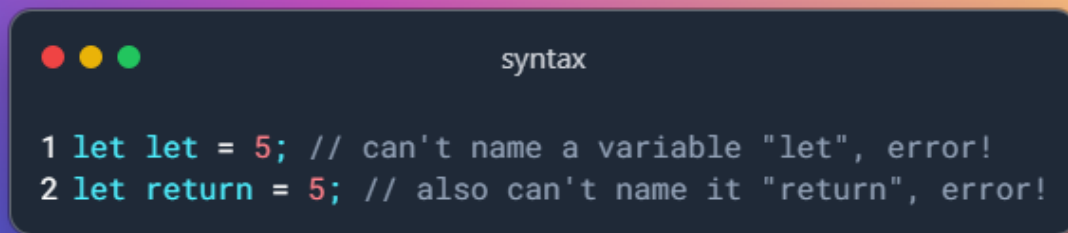
Case matters

Variables named apple and APPLE are two different variables.

There is a list of reserved words, which cannot be used as variable names because they are used by the language itself.

For example: let, class, return, and function are reserved.

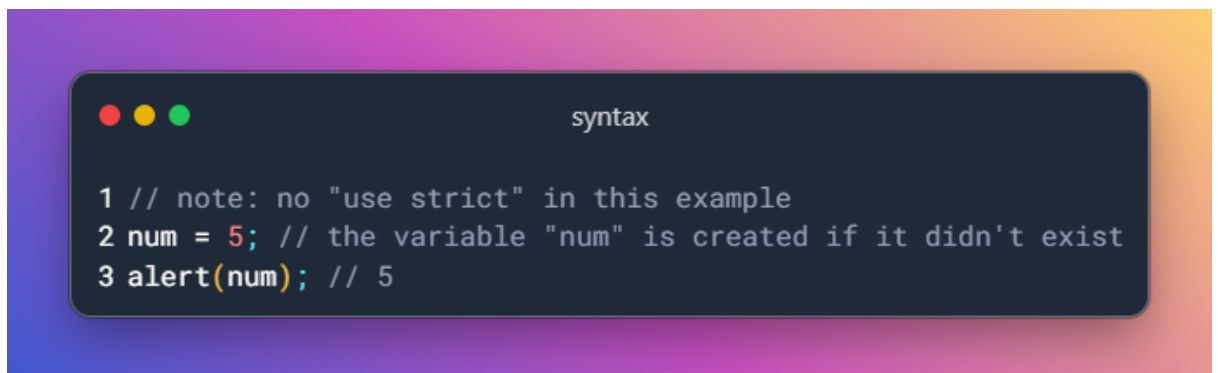
The code below gives a syntax error:



```
1 let let = 5; // can't name a variable "let", error!  
2 let return = 5; // also can't name it "return", error!
```

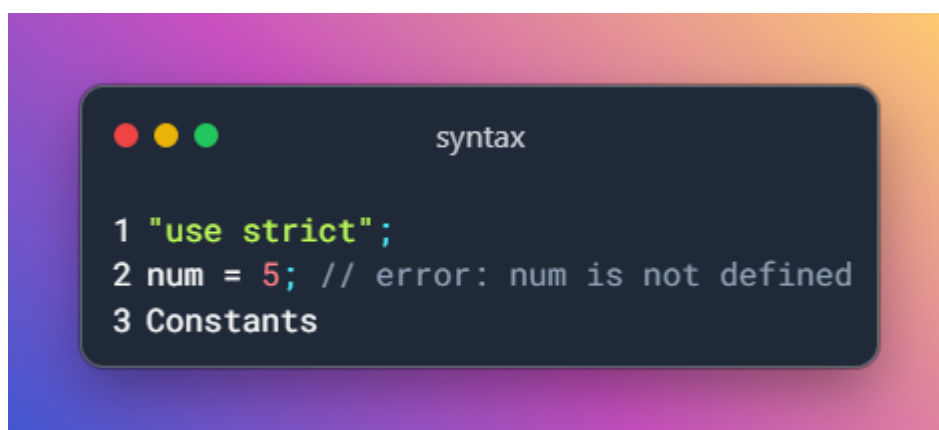
An assignment without use strict

Normally, we need to define a variable before using it. But in the old times, it was technically possible to create a variable by a mere assignment of the value without using let. This still works now if we don't put use strict in our scripts to maintain compatibility with old scripts.



```
1 // note: no "use strict" in this example  
2 num = 5; // the variable "num" is created if it didn't exist  
3 alert(num); // 5
```

This is a bad practice and would cause an error in strict mode:

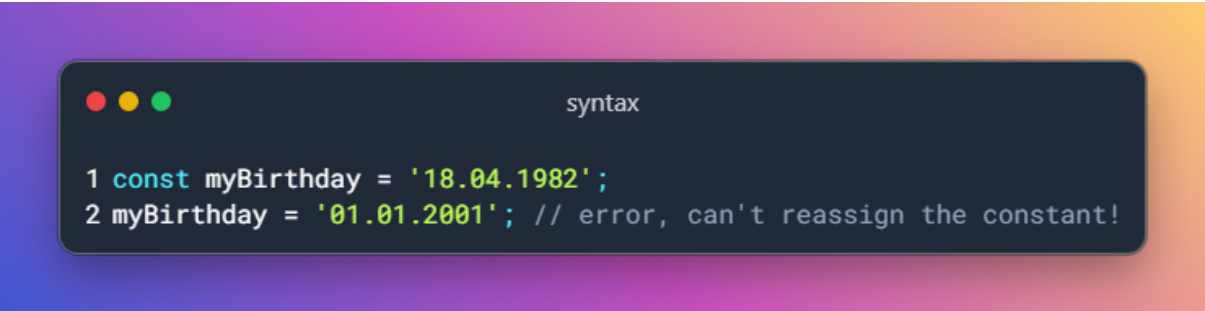


```
1 "use strict";  
2 num = 5; // error: num is not defined  
3 Constants
```

To declare a constant (unchanging) variable, use const instead of let:


```
const myBirthday = '18.04.1982';
```

Variables declared using `const` are called “constants”. They cannot be reassigned. An attempt to do so would cause an error:



```
1 const myBirthday = '18.04.1982';  
2 myBirthday = '01.01.2001'; // error, can't reassign the constant!
```

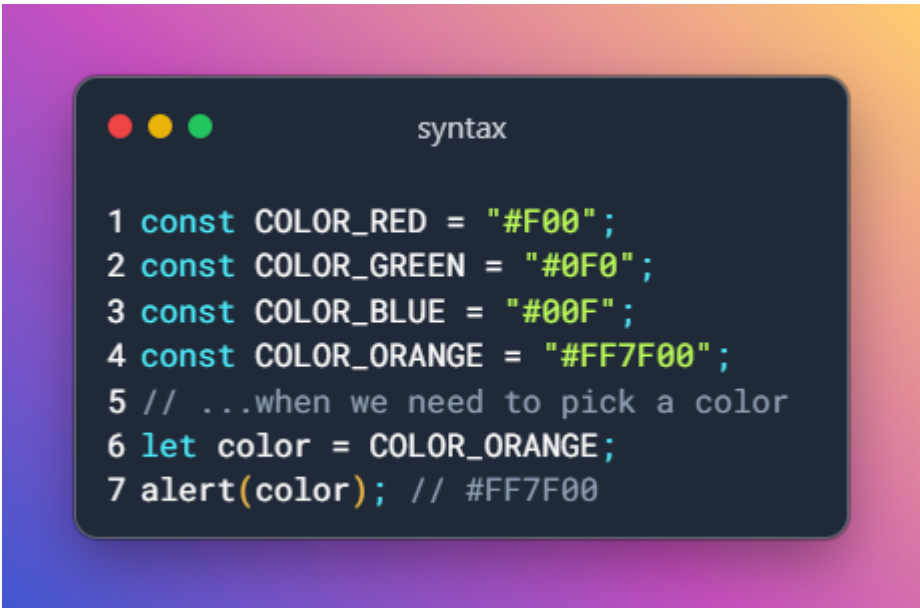
When a programmer is sure that a variable will never change, they can declare it with `const` to guarantee and clearly communicate that fact to everyone.

Uppercase constants

There is a widespread practice to use constants as aliases for difficult-to-remember values that are known prior to execution.

Such constants are named using capital letters and underscores.

For instance, let's make constants for colors in so-called “web” (hexadecimal) format:



```
1 const COLOR_RED = "#F00";  
2 const COLOR_GREEN = "#0F0";  
3 const COLOR_BLUE = "#00F";  
4 const COLOR_ORANGE = "#FF7F00";  
5 // ...when we need to pick a color  
6 let color = COLOR_ORANGE;  
7 alert(color); // #FF7F00
```

Benefits:

`COLOR_ORANGE` is much easier to remember than `"#FF7F00"`.
It is much easier to mistype `"#FF7F00"` than `COLOR_ORANGE`.
When reading the code, `COLOR_ORANGE` is much more meaningful than `#FF7F00`.
When should we use capitals for a constant and when should we name it normally? Let's make that clear.

Being a "constant" just means that a variable's value never changes. But there are constants that are known prior to execution (like a hexadecimal value for red) and there are constants that are calculated in run-time, during the execution, but do not change after their initial assignment.

For instance:

```
const pageLoadTime = /* time taken by a webpage to load */;
```

The value of `pageLoadTime` is not known prior to the page load, so it's named normally. But it's still a constant because it doesn't change after assignment.

In other words, capital-named constants are only used as aliases for "hard-coded" values.

Name things right

Talking about variables, there's one more extremely important thing.

A variable name should have a clean, obvious meaning, describing the data that it stores.

Variable naming is one of the most important and complex skills in programming. A quick glance at variable names can reveal which code was written by a beginner versus an experienced developer.

In a real project, most of the time is spent modifying and extending an existing code base rather than writing something completely separate from scratch. When we return to some code after doing something else for a while, it's much easier to find information that is well-labeled. Or, in other words, when the variables have good names.

Please spend time thinking about the right name for a variable before declaring it. Doing so will repay you handsomely.

Some good-to-follow rules are:

Use human-readable names like `userName` or `shoppingCart`.

Stay away from abbreviations or short names like `a`, `b`, `c`, unless you really know what you're doing.

Make names maximally descriptive and concise. Examples of bad names are `data` and `value`. Such names say nothing. It's only okay to use them if the context of the code makes it exceptionally obvious which data or value the variable is referencing.

Agree on terms within your team and in your own mind. If a site visitor is called a "user" then we should name related variables `currentUser` or `newUser` instead of `currentVisitor` or `newManInTown`.

Data types