HYPERMEDIA APPLICATIONS (WEB AND MULTIMEDIA)

JavaScript Basics

What is JS?

JavaScript is a cross-platform, object-oriented scripting language used to make webpages interactive.

JavaScript enables you to create dynamically updating content, control multimedia, animate images, …

This means that in the browser, JavaScript can change the way the webpage (DOM) looks.

Interpreted versus compiled code

In **interpreted languages**, the code is run from top to bottom and the result of running the code is immediately returned. You don't have to transform the code into a different form before the browser runs it.

Compiled languages on the other hand are transformed (compiled) into another form before they are run by the computer.

JavaScript is an interpreted language run inside the browser.

JS Basics

JavaScript borrows most of its syntax from Java, C, and C++, but it has also been influenced by Awk, Perl, and Python.

JavaScript is **case-sensitive** and uses the **Unicode** character set.

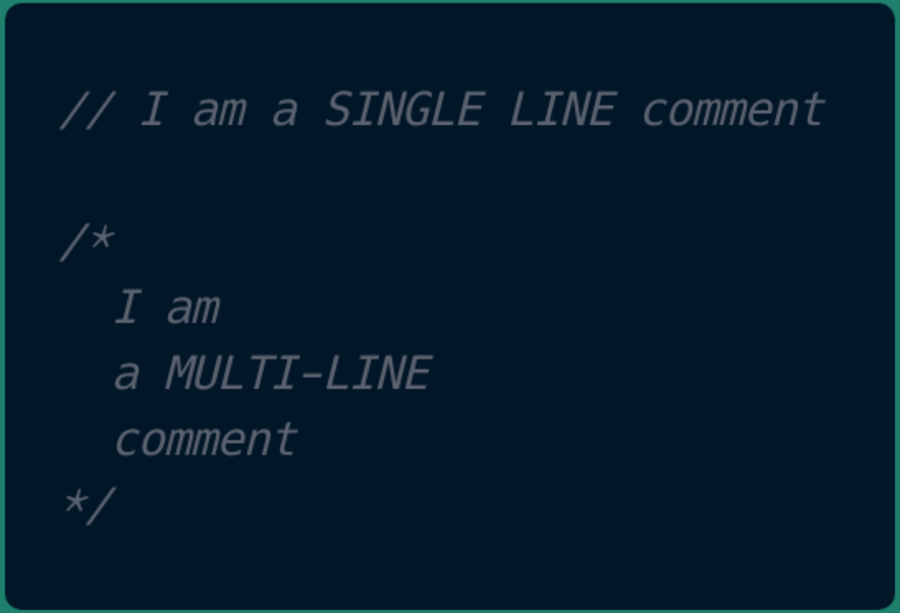
In JavaScript, instructions are called statements and are separated by semicolons (;).

A semicolon is not necessary after a statement if it is written on its own line. But if more than one statement on a line is desired, then they *must* be separated by semicolons.

It is considered best practice, however, to always write a semicolon after a statement, even when it is not strictly needed.

Comments

As with HTML and CSS, it is possible to write comments into your JavaScript code that will be ignored by the browser



```
// I am a SINGLE LINE comment

/*
  I am
  a MULTI-LINE
  comment
*/
```

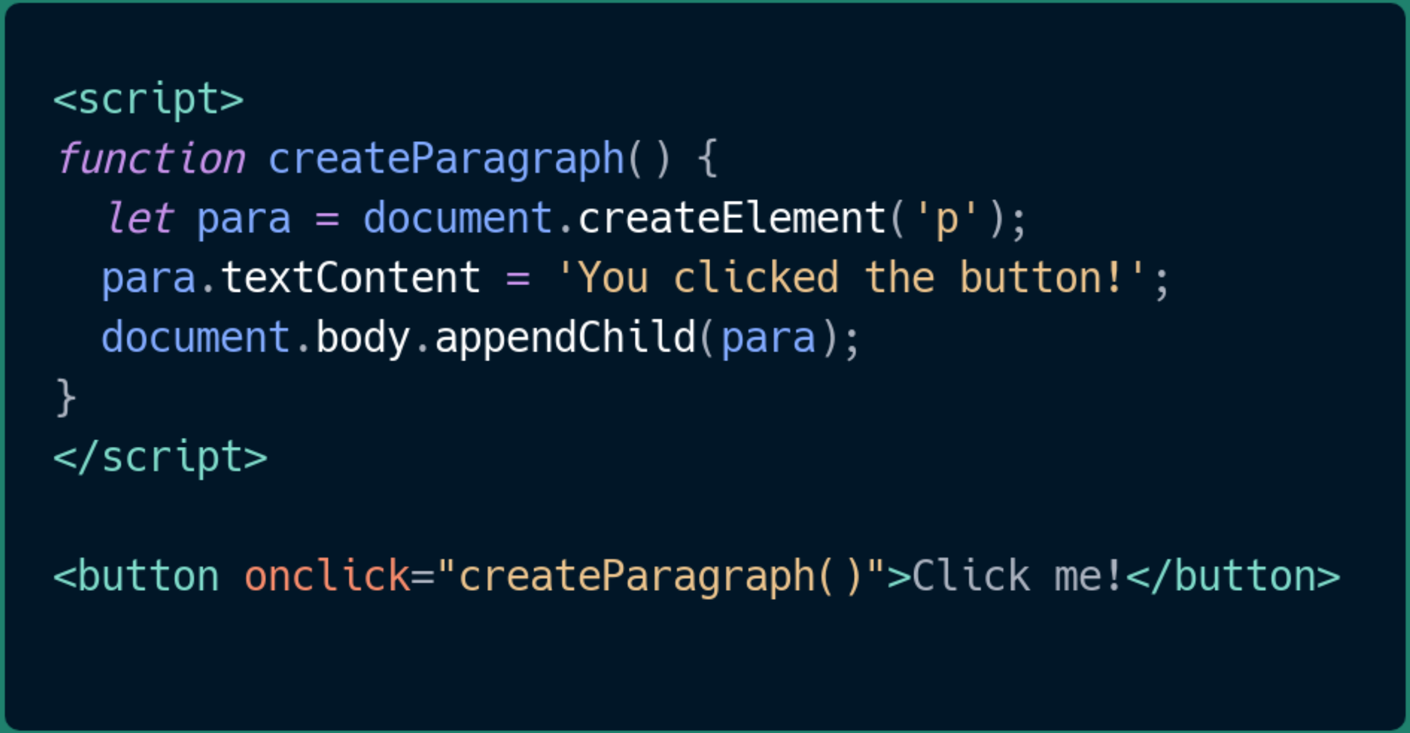
Add JavaScript to your page – Internal JavaScript

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My test page</title>

    <script>

        document.addEventListener("DOMContentLoaded", function() {
        function createParagraph() {
            let para = document.createElement('p');
            para.textContent = 'You clicked the button!';
            document.body.appendChild(para);
        }

        const buttons = document.querySelectorAll('button');

        for(let i = 0; i < buttons.length ; i++) {
            buttons[i].addEventListener('click', createParagraph);
        }
    });

    </script>

  </head>
  <body>
    <p>This is my page</p>
  </body>
</html>
```

Add JavaScript to your page – External JavaScript

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My test page</title>

    <script src="script.js" defer></script>

  </head>
  <body>
    <p>This is my page</p>
  </body>
</html>
```

Add JavaScript to your page – Inline JavaScript



```
<script>
function createParagraph() {
  let para = document.createElement('p');
  para.textContent = 'You clicked the button!';
  document.body.appendChild(para);
}
</script>

<button onclick="createParagraph()">Click me!</button>
```

async and defer

Scripts loaded using the **async** attribute will download the script without blocking rendering the page and will execute it as soon as the script finishes downloading. (no order)

Scripts loaded using the **defer** attribute will run in the order they appear in the page and execute them as soon as the script and content are downloaded. (order matters)

Errors in JavaScript

Syntax errors: These are spelling errors in your code that actually cause the program not to run at all, or stop working part way through — you will usually be provided with some error messages too.

Logic errors: These are errors where the syntax is actually correct, but the code is not what you intended it to be, meaning that program runs successfully but gives incorrect results.

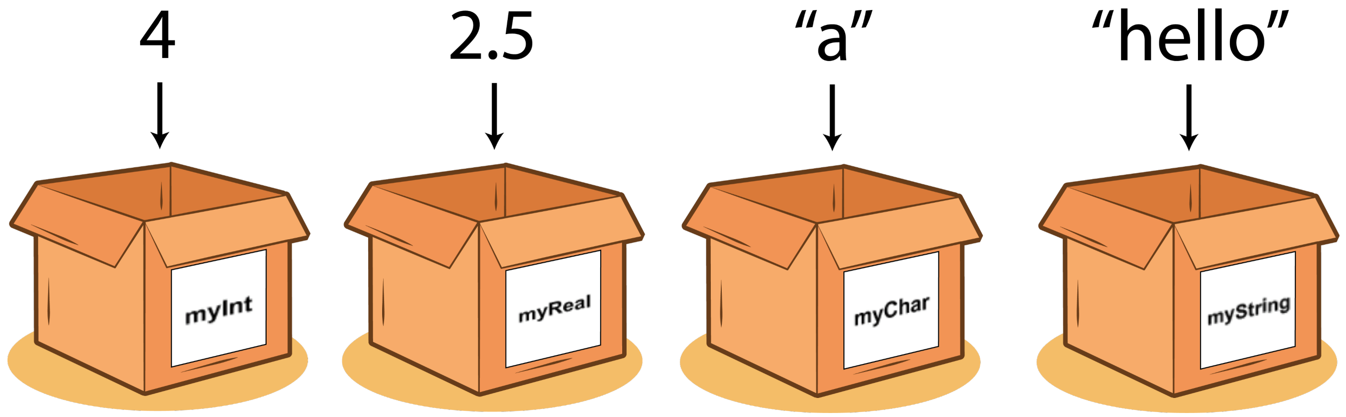
In Javascript you can have both, so be careful!!

What is a variable?

A variable is a container for a value (number, string, complex data, functions).

One special thing about variables is that their contained values can change.

Note: Variables contain values. Variables aren't the values themselves; they are containers for values.

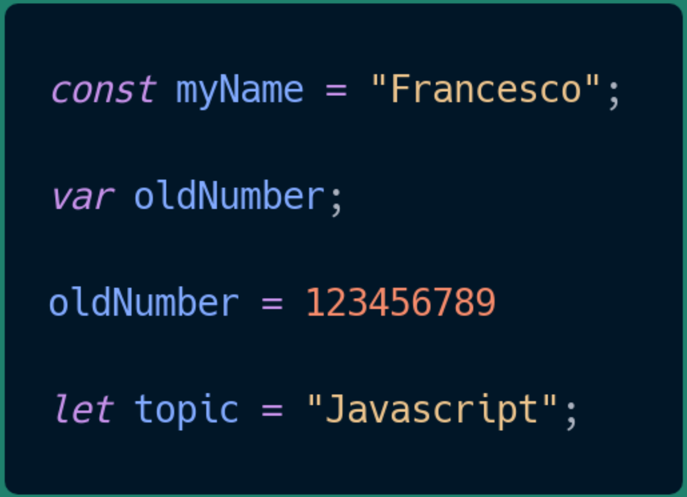


Declaring a variable

To use a variable, you've first got to declare it. To do this, we type the keyword var, let or const followed by the name you want to call your variable.

Once you've declared a variable, you can initialize it with a value. You do this by typing the variable name, followed by an equals sign (=), followed by the value you want to give it.

Once a variable has been initialized with a value, you can update it by giving it a different value.



```
const myName = "Francesco";

var oldNumber;

oldNumber = 123456789

let topic = "Javascript";
```

Declaring a variable

JavaScript has three kinds of variable declarations:

- **var:** Declares a variable, optionally initializing it to a value.
- **let:** Declares a block-scoped, local variable, optionally initializing it to a value.
- **const:** Declares a block-scoped, read-only named constant that must be initialized.

A variable declared using the var or let statement with no assigned value specified has the value of undefined.

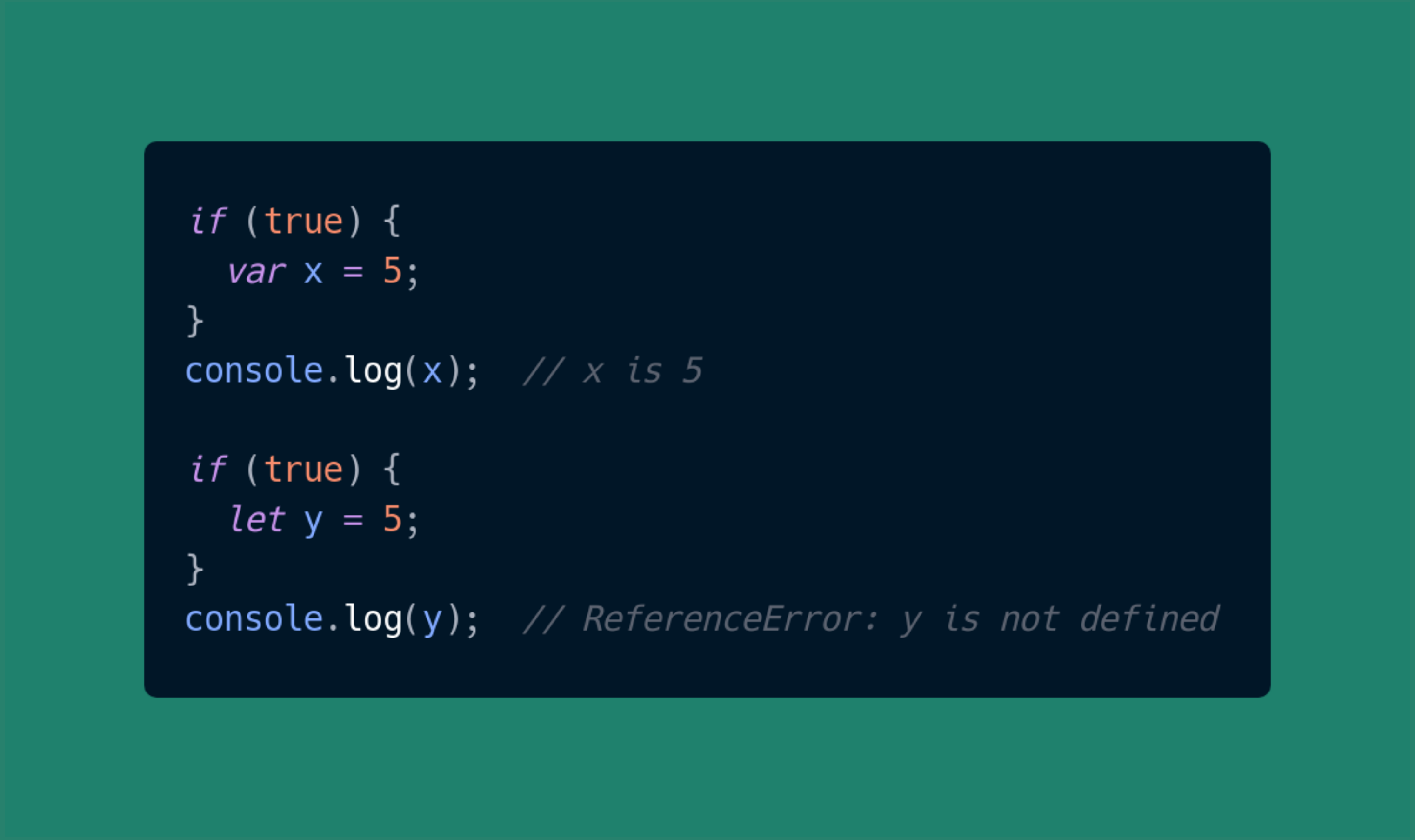
An attempt to access an undeclared variable results in a ReferenceError exception

Variable scope

When you declare a variable outside of any function, it is called a *global* variable, because it is available to any other code in the current document.

When you declare a variable within a function, it is called a *local* variable, because it is available only within that function.

Variable scope



```
if (true) {
  var x = 5;
}
console.log(x);  // x is 5

if (true) {
  let y = 5;
}
console.log(y);  // ReferenceError: y is not defined
```

Variable hoisting

Another unusual thing about variables in JavaScript is that you can refer to a variable declared later, without getting an exception.

This concept is known as **hoisting.**

However, variables that are hoisted return a value of undefined.

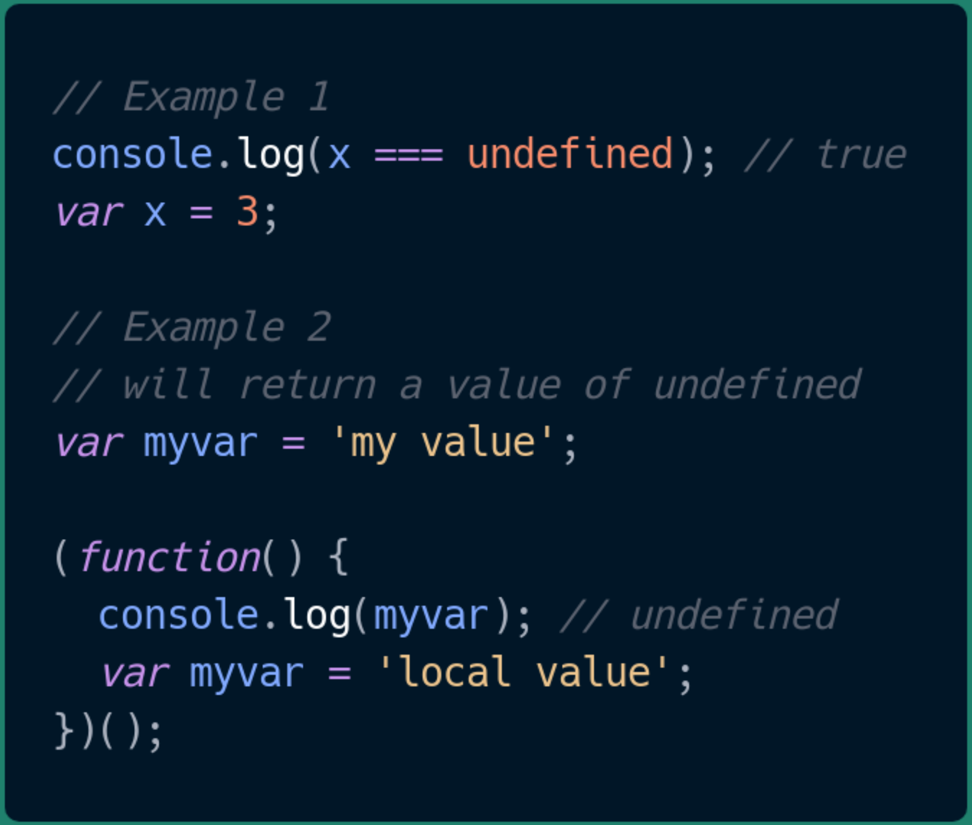
So even if you declare and initialize a variable after you use it or refer to it, it still returns undefined.

Because of hoisting, all var statements in a function should be placed as near to the top of the function as possible.

This best practice increases the clarity of the code.

In ECMAScript 2015, let and const **are hoisted but not initialized**.

Variable hoisting



```
// Example 1
console.log(x === undefined); // true
var x = 3;

// Example 2
// will return a value of undefined
var myvar = 'my value';

(function() {
  console.log(myvar); // undefined
  var myvar = 'local value';
})();
```

Variable types

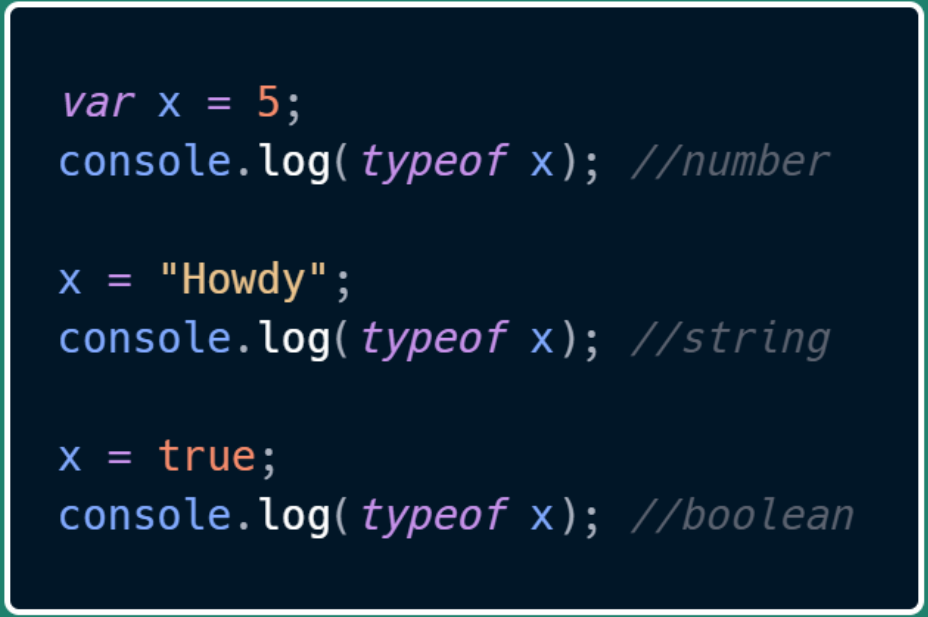
The latest ECMAScript standard defines eight data types:

- Seven data types that are primitives:
  - **Boolean**: true and false.
  - **null**: a special keyword denoting a null value.
    (Because JavaScript is case-sensitive, null is not the same as Null, NULL, or any other variant.)
  - **undefined**: a top-level property whose value is not defined.
  - **Number**: an integer or floating point number. For example: 42 or 3.14159.
  - **BigInt**: an integer with arbitrary precision. For example: 9007199254740992n.
  - **String**: a sequence of characters that represent a text value.
    For example: "Howdy"
  - **Symbol** (new in ECMAScript 2015): a data type whose instances are unique and immutable.
- and **Object**

Data type conversion

JavaScript is a "dynamically typed language" which means that, unlike some other languages, you don't need to specify what data type a variable will contain (numbers, strings, arrays, etc).

It also means that data types are automatically converted as-needed during script execution.



```
var x = 5;
console.log( typeof x); //number

x = "Howdy";
console.log( typeof x); //string

x = true;
console.log( typeof x); //boolean
```

Operators

JavaScript has both *binary* and *unary* operators, and one special ternary operator (the conditional operator).

A binary operator requires two operands, one before the operator and one after the operator

- Arithmetic operators (+, -, %, *, /)
- Increment and decrement operators (++, --)
- Assignment operators (=, +=, -= …)
- Comparison operators
- Logical operators
- Ternary operators

Logical operator

- && — AND : returns true if both operands are true; otherwise, returns false.
- || — OR : returns true if either operand is true; if both are false, returns false.
- ! — NOT : returns false if its single operand that can be converted to true; otherwise, returns true.

Comparison operator

- === and !== — returns true if the operands are (not) equal and of the same type.
- < and > — returns true if one value is less than or greater than another.
- <= and >= — returns true if one value is less than or equal to, or greater than or equal to, another.
- == and != — return true if the operands are (not) equal.

More on the difference between the == and === operators later on, since they are specific for JS.

Ternary operator

The conditional operator is the only JavaScript operator that takes three operands.

The operator can have one of two values based on a condition.

condition ? val1 : val2

If condition is true, the operator has the value of val1. Otherwise it has the value of val2.

You can use the conditional operator anywhere you would use a standard operator.

Conditional operator

```
let greeting = ( isBirthday ) ? 'Happy birthday Mrs. Smith — we hope you
have a great day!' : 'Good morning Mrs. Smith.';
```

Types of numbers

- **Integers** are whole numbers, e.g. 10, 400, or -5.
- **Floating point numbers** (floats) have decimal points and decimal places, for example 12.5, and 56.7786543.
- **Doubles** are a specific type of floating point number that have greater precision than standard floating point numbers (meaning that they are accurate to a greater number of decimal places).

Despite those differences, JavaScript treats all the numbers as 64-bit Floating Points.

Number Object

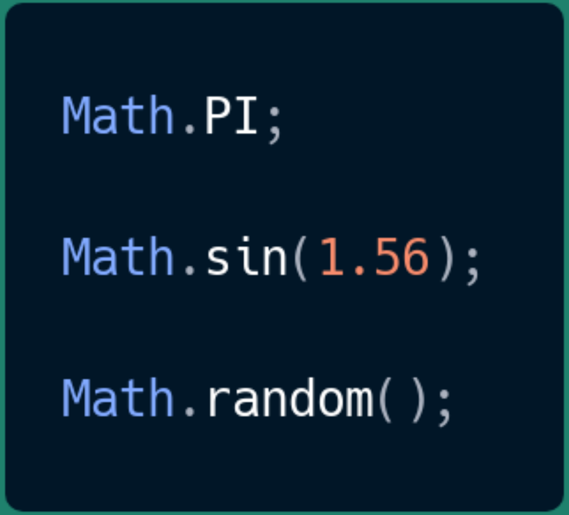
The built-in Number object has properties for numerical constants, such as maximum value, not-a-number, and infinity.

You cannot change the values of these properties.

```javascript
var biggestNum = Number.MAX_VALUE;
var smallestNum = Number.MIN_VALUE;
var infiniteNum = Number.POSITIVE_INFINITY;
var negInfiniteNum = Number.NEGATIVE_INFINITY;
var notANum = Number.NaN;
```

Math Object

The built-in Math object has properties and methods for mathematical constants and functions.

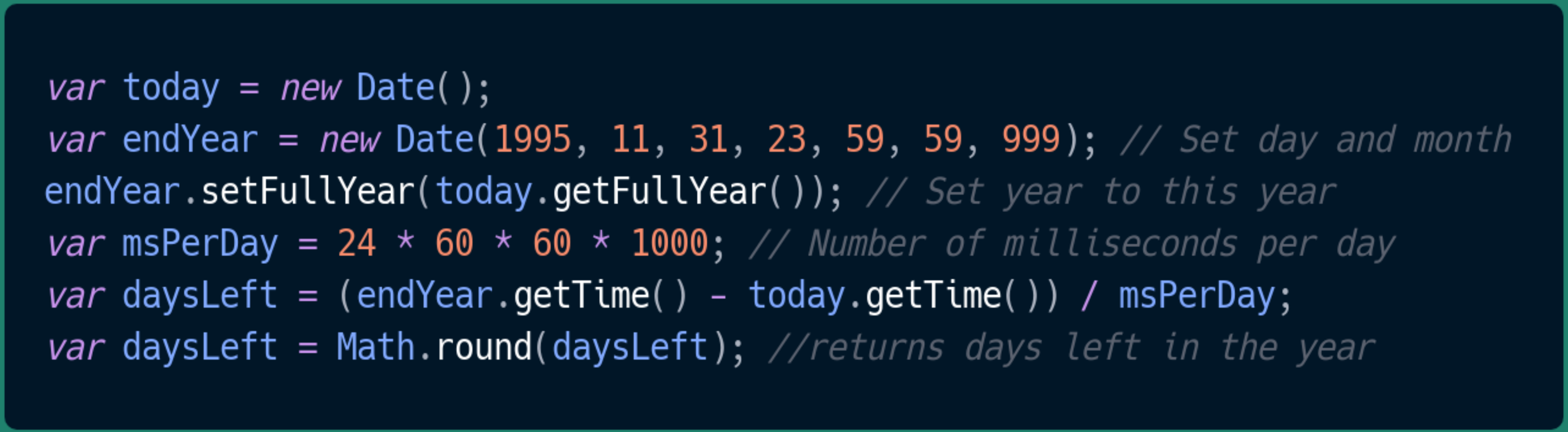


```
Math.PI;

Math.sin(1.56);

Math.random( );
```

Date Object

JavaScript does not have a date data type.

However, you can use the Date object and its methods to work with dates and times in your applications.



```javascript
var today = new Date();
var endYear = new Date(1995, 11, 31, 23, 59, 59, 999); // Set day and month
endYear.setFullYear(today.getFullYear()); // Set year to this year
var msPerDay = 24 * 60 * 60 * 1000; // Number of milliseconds per day
var daysLeft = (endYear.getTime() - today.getTime()) / msPerDay;
var daysLeft = Math.round(daysLeft); //returns days left in the year
```

String

A string stores a series of characters like "John Doe".

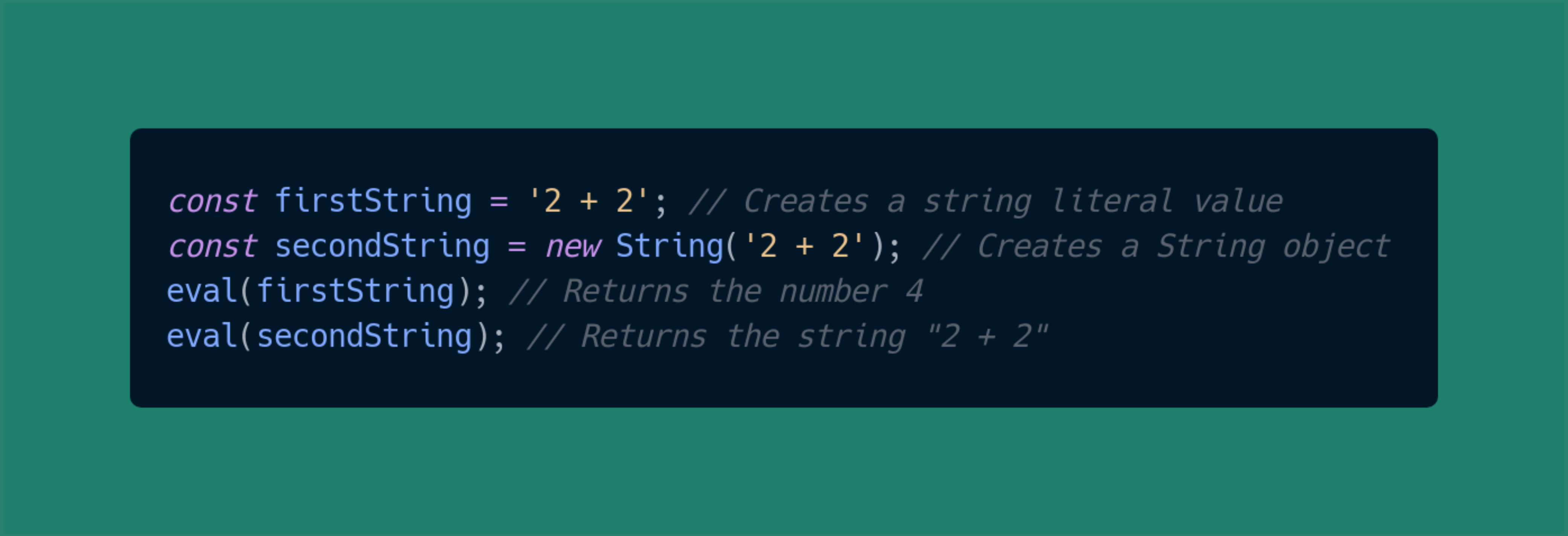
A string can be any text inside double or single quotes. You can also use the less-known backtick (`) in addition to the previous two.

In the end there are three ways of creating a string:

- Using string literals
- Using string object
- Using template literals

String object

You should use string literals unless you specifically need to use a String object, because String objects can have counterintuitive behavior.



```javascript
const firstString = '2 + 2'; // Creates a string literal value
const secondString = new String('2 + 2'); // Creates a String object
eval(firstString); // Returns the number 4
eval(secondString); // Returns the string "2 + 2"
```

Template literals

Template literals are another type of string syntax that provides more flexible, easier to read strings.

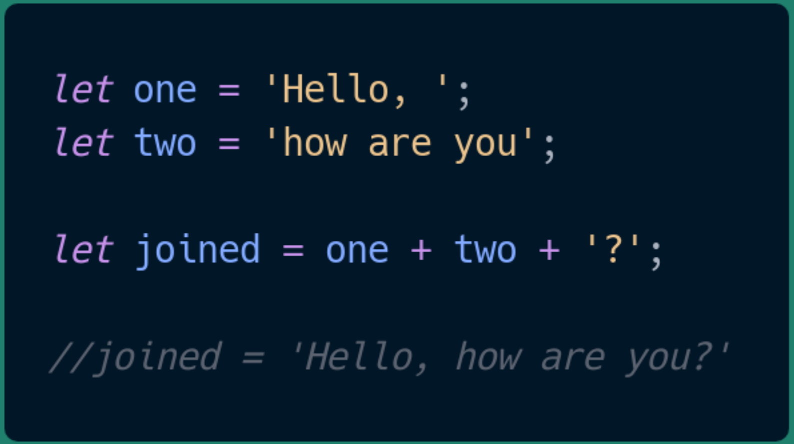
To turn a standard string literal into a template literal, you have to replace the quote marks (' ', or " ") with backtick characters (` `).

Template literals can contain place holders. These are indicated by the Dollar sign and curly braces (${expression}).

```
let song = 'Fight the Youth’;

song = `Fight the Youth`;

let score = 9;
let highestScore = 10;

//If we want to concatenate strings, or include expression results inside
them, traditional strings can //be fiddly to write:

let output = 'I like the song "' + song + '". I gave it a score of ' +
(score/highestScore * 100) + '%.';

output = `I like the song "${ song }". I gave it a score of ${
score/highestScore * 100 }%.`;
```

Concatenation

Concatenate uses the plus (+) operator, to join two strings.



```
let one = 'Hello, ';
let two = 'how are you';

let joined = one + two + '?';

//joined = 'Hello, how are you?'
```

String methods and properties

Finding the length of a string: the length property.

Retrieving a specific string character: you can return any character inside a string by using **square bracket notation** — this means you include square brackets ([]) on the end of your variable name. Inside the square brackets you include the number of the character you want to return (starting from 0)

Updating parts of a string: You can replace one substring inside a string with another substring using the replace() method. It takes two parameters — the string you want to replace, and the string you want to replace it with.

Changing case: The string methods toLowerCase() and toUpperCase() take a string and convert all the characters to lower- or uppercase, respectively. This can be useful for example if you want to normalize all user-entered data before storing it in a database.

CODEPEN

Strings

Arrays

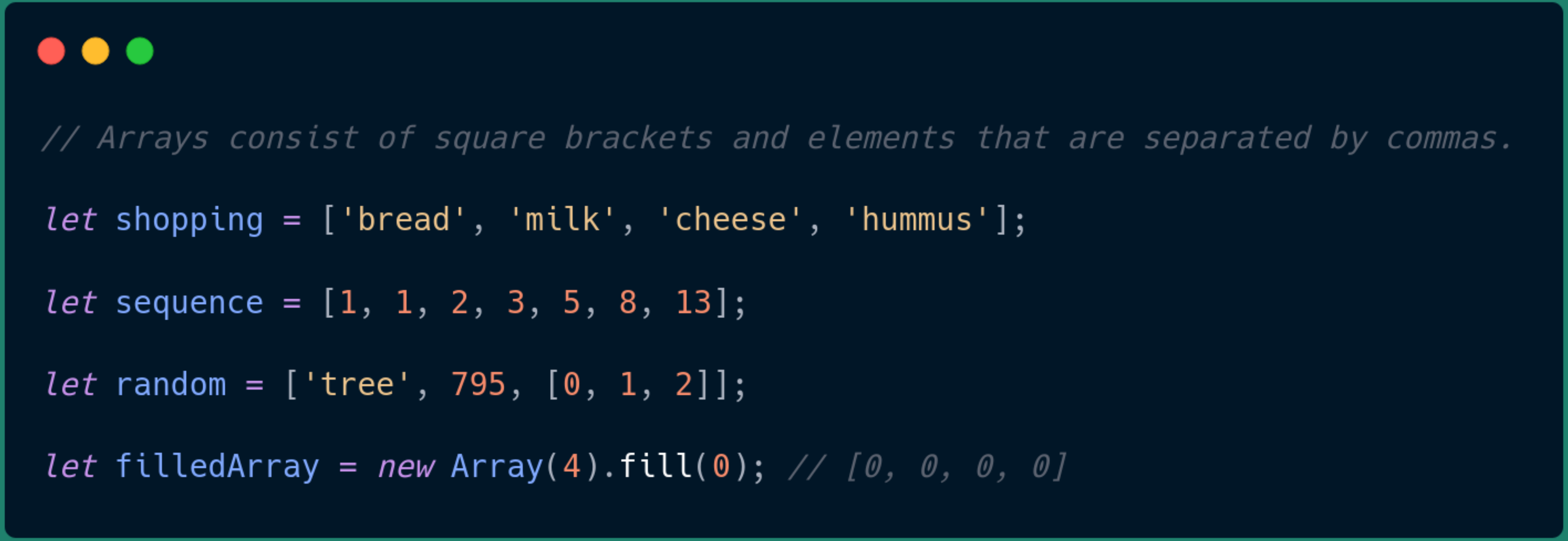
An *array* is an ordered list of values that you refer to with a name and an index.

JavaScript does not have an explicit array data type.

You can use the predefined Array object and its methods to work with arrays in your applications.

The Array object has methods for manipulating arrays in various ways, such as joining, reversing, and sorting them.
It has a property for determining the array length and other properties for use with regular expressions.

Creating arrays



```javascript
// Arrays consist of square brackets and elements that are separated by commas.

let shopping = ['bread', 'milk', 'cheese', 'hummus'];

let sequence = [1, 1, 2, 3, 5, 8, 13];

let random = ['tree', 795, [0, 1, 2]];

let filledArray = new Array(4).fill(0); // [0, 0, 0, 0]
```

Accessing and modifying array items

```
//You can then access individual items in the array using bracket notation,
in the same way that you accessed the letters in a string.

let shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];

shopping[0]; // returns "bread"

// You can also modify an item in an array by giving a single array item a
new value. Try this:

shopping[0] = 'tahini';
// shopping will now return [ "tahini", "milk", "cheese", "hummus",
"noodles" ]
```

Converting between strings and arrays

```
// from string to array
let myData = 'Manchester,London,Liverpool,Birmingham,Leeds,Carlisle';

let myArray = myData.split(',');
myArray; //myArray = ["Manchester", "London", "Liverpool", "Birmingham",
"Leeds", "Carlisle"]

//from array to string

let myNewString = myArray.join(',');
myNewString; // "Manchester,London,Liverpool,Birmingham,Leeds,Carlisle"

let dogNames = ['Rocket','Flash','Bella','Slugger'];
dogNames.toString(); // Rocket,Flash,Bella,Slugger
```

Adding and removing array items

```
let myArray = ['Manchester', 'London', 'Liverpool', 'Birmingham', 'Leeds',
'Carlisle'];

myArray.push('Cardiff');
//myArray = ['Manchester', 'London', 'Liverpool', 'Birmingham', 'Leeds',
'Carlisle', 'Cardiff']

let removedItem = myArray.pop();
myArray; // ['Manchester', 'London', 'Liverpool', 'Birmingham', 'Leeds',
'Carlisle']
removedItem; //Cardiff
```

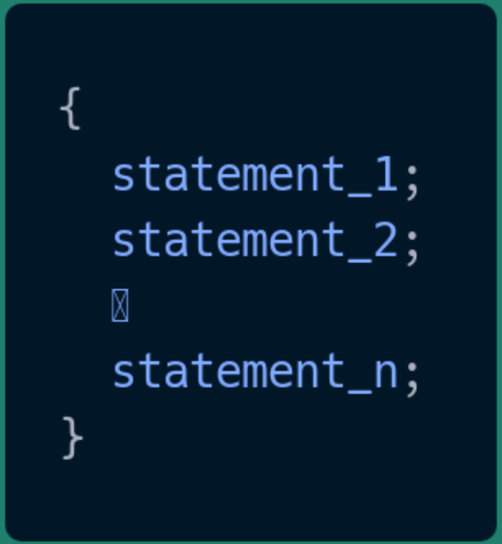
Adding and removing array items

```
// unshift() and shift() work in exactly the same way as push() and pop(),
respectively, except that they work on the beginning of the array, not the
end.

myArray.unshift('Edinburgh');
myArray;


let removedItem = myArray.shift();
myArray;
removedItem;
```

Block statement

The most basic statement is a ***block statement***, which is used to group statements. The block is delimited by a pair of curly brackets.

Block statements are commonly used with control flow statements (if, for, while).

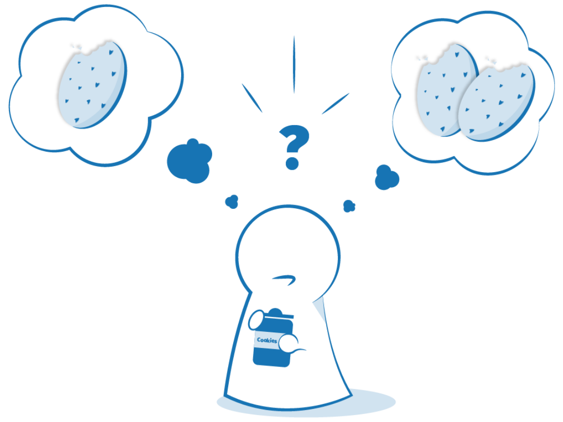


```
{
  statement_1;
  statement_2;
  …
  statement_n;
}
```

Conditional statements

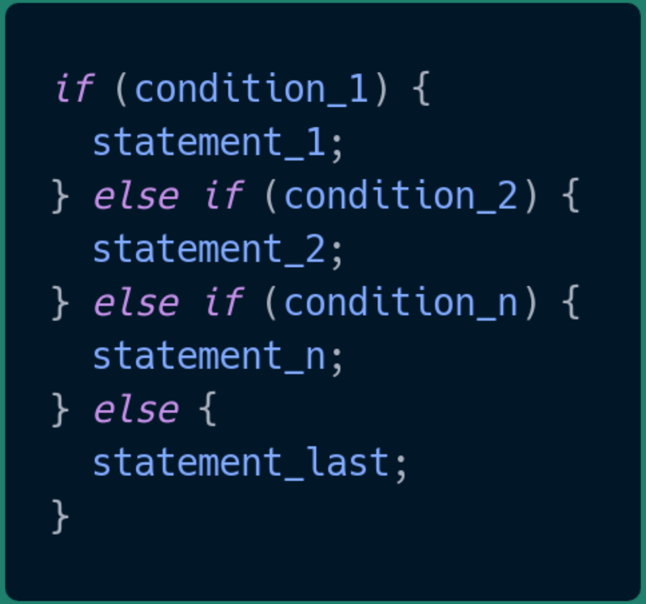
A conditional statement is a set of commands that executes if a specified condition is true.

JavaScript supports two conditional statements: if...else and switch.



If...else statement

Use the if statement to execute a statement if a logical condition is true. Use the optional else clause to execute a statement if the condition is false.



```
if (condition_1) {
  statement_1;
} else if (condition_2) {
  statement_2;
} else if (condition_n) {
  statement_n;
} else {
  statement_last;
}
```

Falsy values

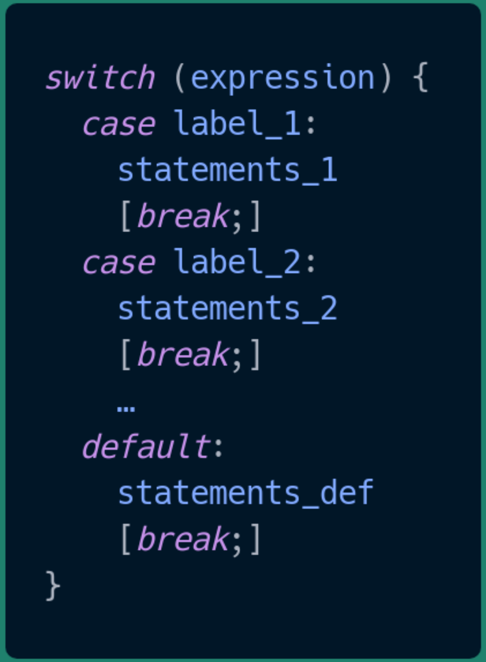
The following values evaluate to false:

- false
- undefined
- null
- 0
- NaN
- the empty string ("")

All other values—including all objects—evaluate to true when passed to a conditional statement.

Switch statement

A switch statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement.



```
switch (expression) {
  case label_1:
    statements_1
    [break;]
  case label_2:
    statements_2
    [break;]
    …
  default:
    statements_def
    [break;]
}
```

Switch statement

JavaScript evaluates the switch statement as follows:

- The program first looks for a case clause with a label matching the value of expression and then transfers control to that clause, executing the associated statements.
- If no matching label is found, the program looks for the optional default clause:
  - If a default clause is found, the program transfers control to that clause, executing the associated statements.
  - If no default clause is found, the program resumes execution at the statement following the end of switch.

CODEPEN

Conditions

Loops and iteration

Programming loops are all to do with doing the same thing over and over again, which is termed **iteration**.

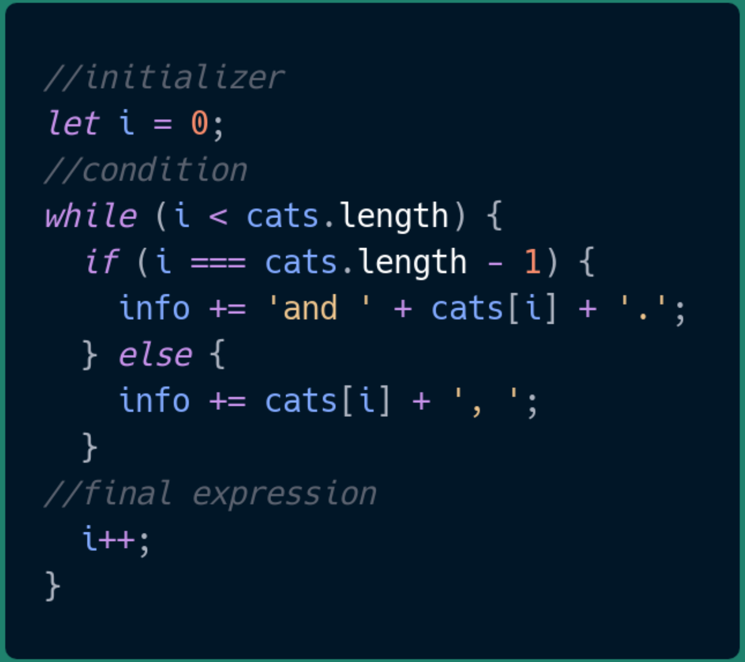
The logical components of a loop are:

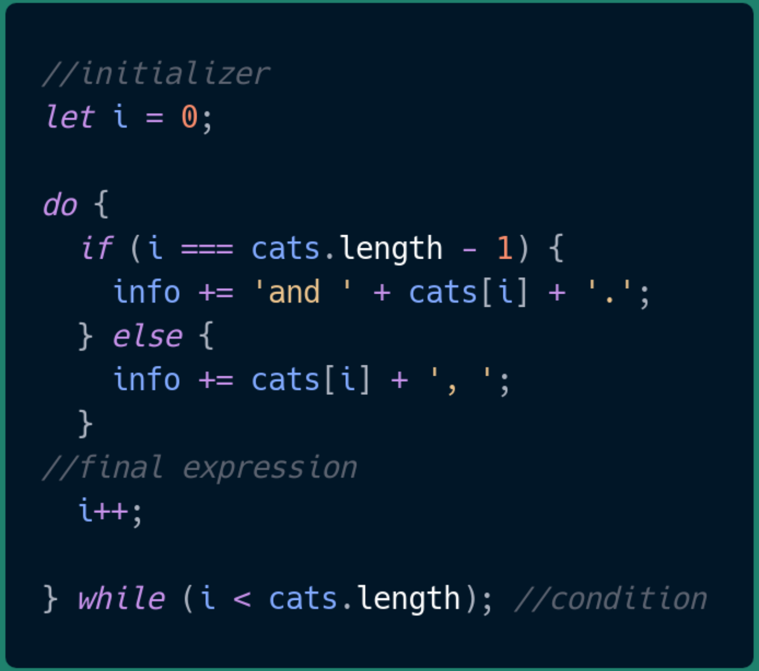
- A **counter**, which is initialized with a certain value — this is the starting point of the loop.
- A **condition**, which is a true/false test to determine whether the loop continues to run, or stops.
- An **iterator**, which generally increments the counter by a small amount on each successive loop until the condition is no longer true.

for statement

```javascript
const cats = ['Bill', 'Jeff', 'Pete', 'Biggles', 'Jasmin'];
let info = 'My cats are called ';
const para = document.querySelector('p');

for (let i = 0; i < cats.length; i++) {
  info += cats[i] + ', ';
}

para.textContent = info;

//My cats are called Bill, Jeff, Pete, Biggles, Jasmin,
```

while and do...while statements

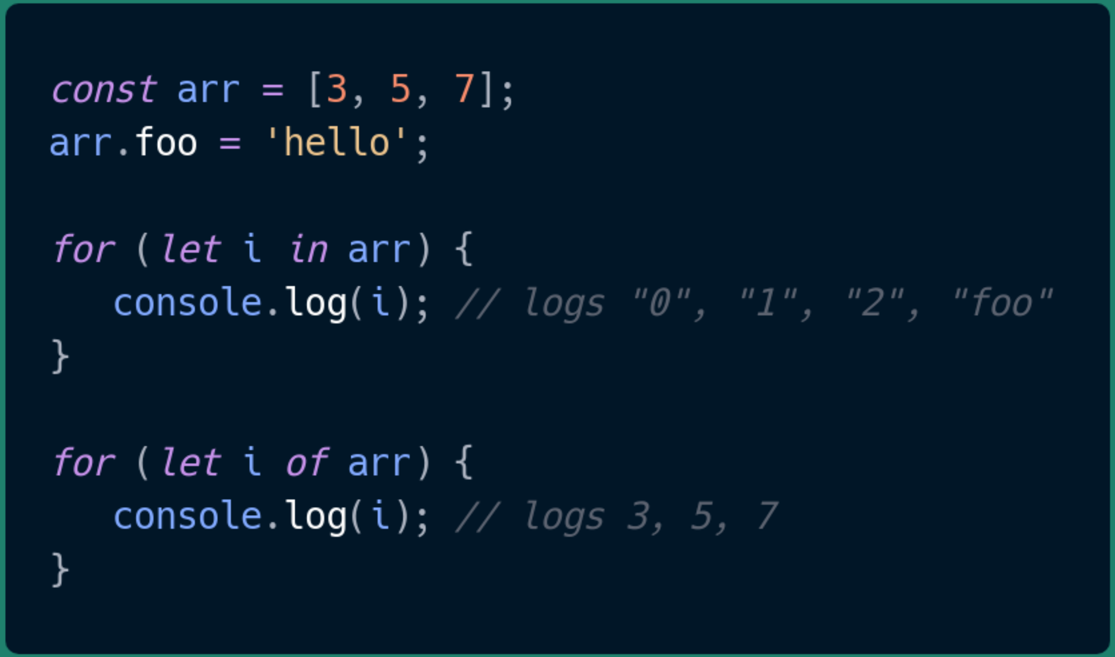

```
//initializer
let i = 0;
//condition
while (i < cats.length) {
  if (i === cats.length - 1) {
    info += 'and ' + cats[i] + '.';
  } else {
    info += cats[i] + ', ';
  }
//final expression
  i++;
}
```


```
//initializer
let i = 0;

do {
  if (i === cats.length - 1) {
    info += 'and ' + cats[i] + '.';
  } else {
    info += cats[i] + ', ';
  }
//final expression
  i++;

} while (i < cats.length); //condition
```

for...of for...in statement

for...of statement creates
a loop Iterating
over iterable
objects (property values)

for...in instead iterates
over property names



```
const arr = [3, 5, 7];
arr.foo = 'hello';

for (let i in arr) {
   console.log(i); // logs "0", "1", "2", "foo"
}

for (let i of arr) {
   console.log(i); // logs 3, 5, 7
}
```

CODEPEN

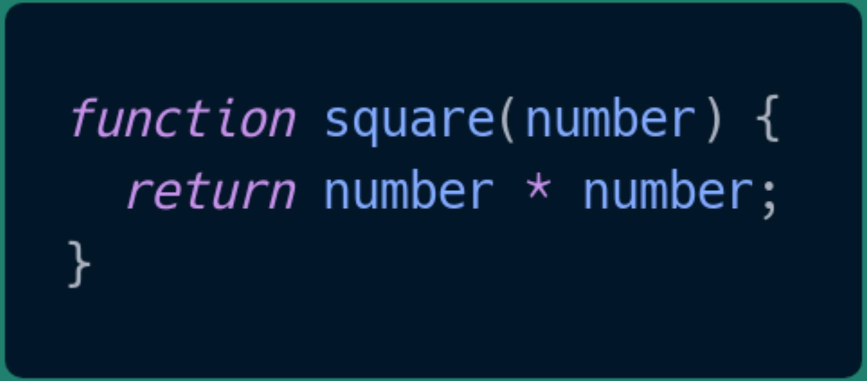
Loops

Functions

A **function definition** (also called a **function declaration**, or **function statement**) consists of the function keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, {...}.

Named function



```
function square(number) {
  return number * number;
}
```

Function expression

Functions can also be created by a function expression.

Such a function can be **anonymous**; it does not have to have a name.

However, a name *can* be provided with a function expression.

Function expressions are convenient when passing a function as an argument to another function.

Function expression

```
const square = function(number) {
  return number * number;
}

var x = square(4) // x gets the value 16
```

```
function map(f, a) {
  let result = []; // Create a new Array
  let i; // Declare variable
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}
const f = function(x) {
   return x * x * x;
}
let numbers = [0, 1, 2, 5, 10];
let cube = map(f,numbers);
console.log(cube);
// Function returns: [0, 1, 8, 125, 1000].
```

Calling functions

Defining a function does not *execute* it. Defining it names the function and specifies what to do when the function is called.

Calling the function actually performs the specified actions with the indicated parameters.

Functions must be *in scope* when they are called, but the function declaration can be hoisted (appear below the call in the code),

Function hoisting only works with function *declarations*—not with function *expressions*.

Function hoisting

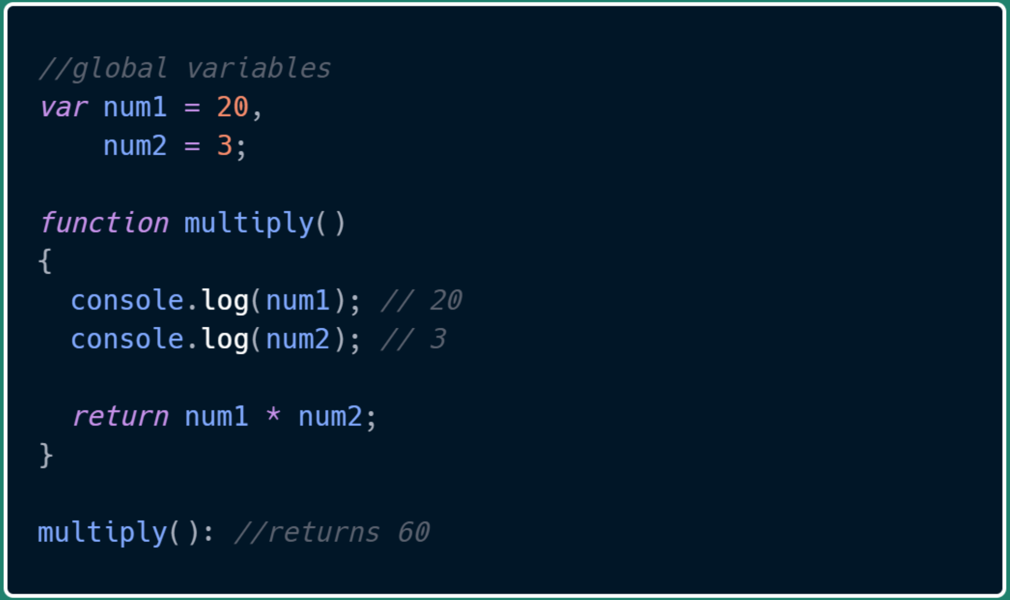
```
// this will work
console.log(square(5));
/* ... */
function square(n) { return n * n }

//this won't work
console.log(square(5)) // Uncaught TypeError: square is not a function
const square = function(n) {
  return n * n;
}
```

Function scope

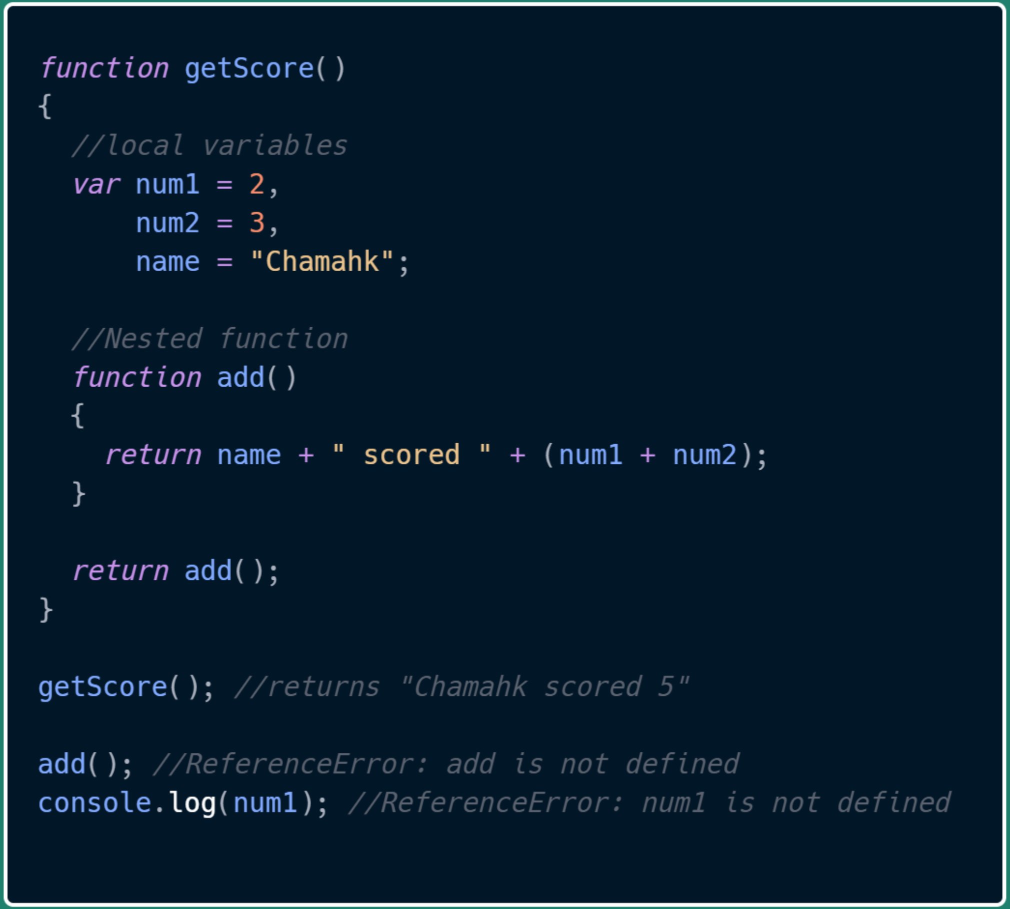
A function defined in the global scope can access all variables defined in the global scope. Moreover, a function defined inside another function can also access all variables defined in its parent function, and any other variables to which the parent function has access to.

On the contrary, variables defined inside a function cannot be accessed from anywhere outside, because the variables are defined only in the scope of the function.

Function scope



```
//global variables
var num1 = 20,
    num2 = 3;

function multiply()
{
  console.log(num1); // 20
  console.log(num2); // 3

  return num1 * num2;
}

multiply(): //returns 60
```



```
function getScore()
{
  //local variables
  var num1 = 2,
      num2 = 3,
      name = "Chamahk";

  //Nested function
  function add()
  {
    return name + " scored " + (num1 + num2);
  }

  return add();
}

getScore(); //returns "Chamahk scored 5"

add(); //ReferenceError: add is not defined
console.log(num1); //ReferenceError: num1 is not defined
```

Closure

During the execution of a function, all of its local variables are created, used and then, at the end, they cease to exist.

There is a way, though, to preserve all of the scope chains of a function and the state of the variables even after it has concluded its execution.

To do this we use closures, a particular kind of nested functions.

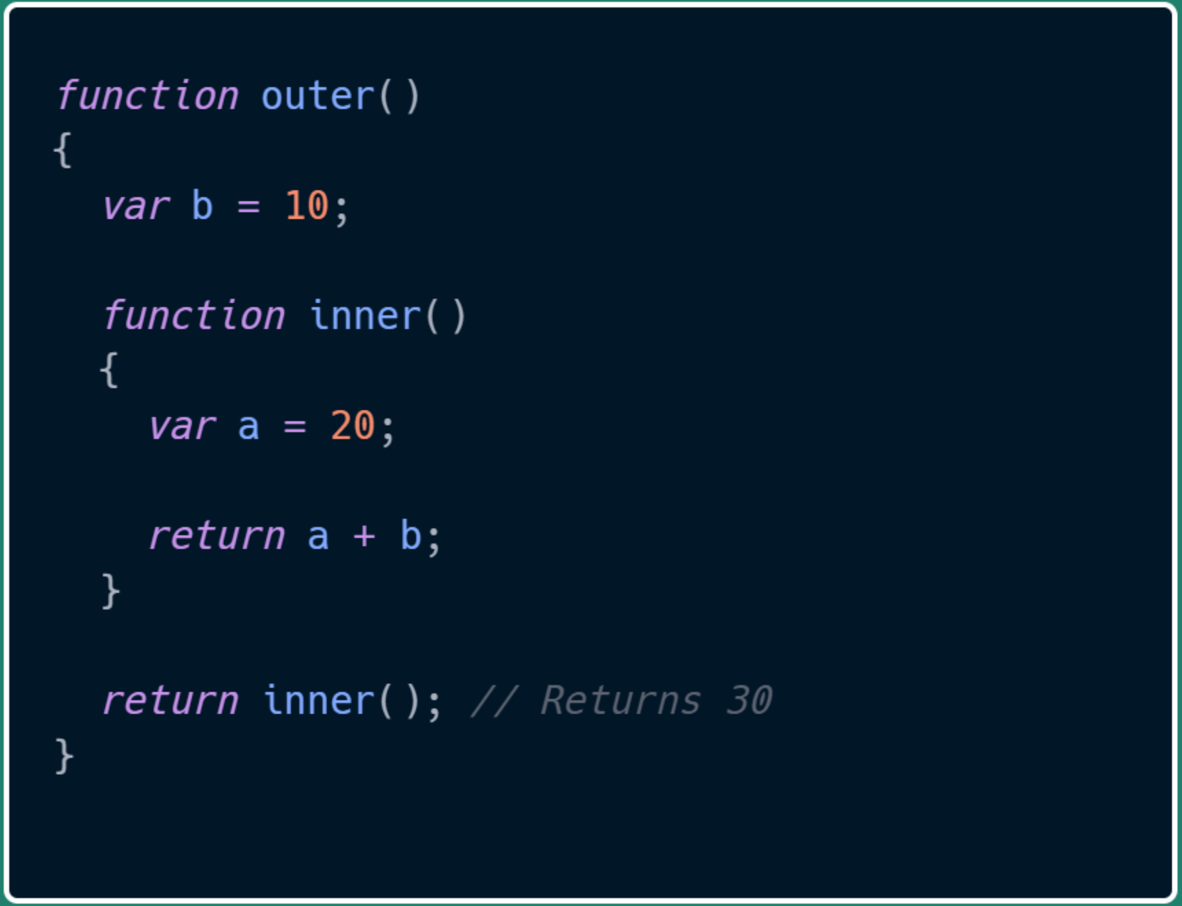
Closures

When the outer function is called its local variable (b) is created.

When the inner function is called its local variable (a) is created.

When the inner function is concluded its local variable (a) ceases to exist.

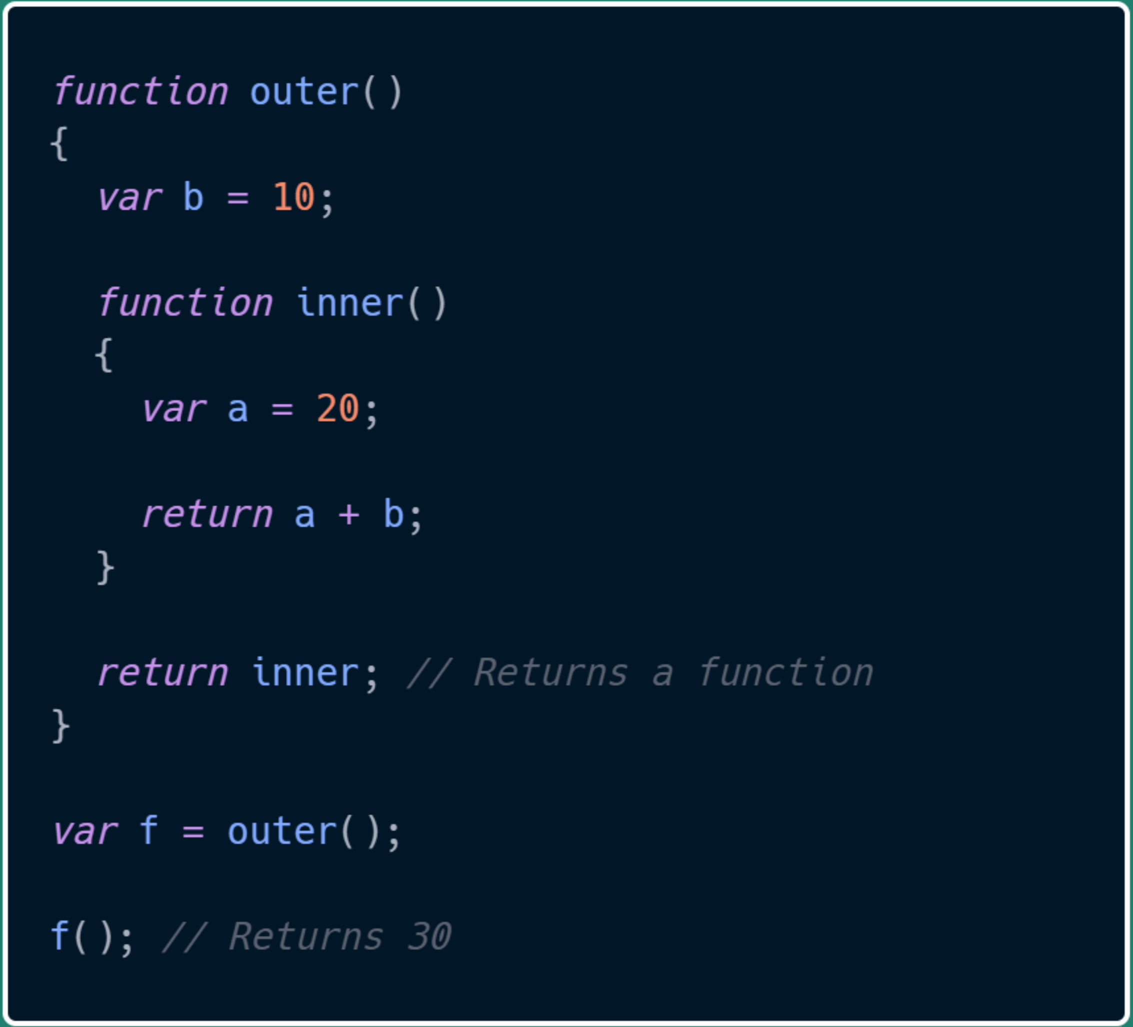
When the outer function is concluded its local variable (b) ceases to exist.



```
function outer()
{
  var b = 10;

  function inner()
  {
    var a = 20;

    return a + b;
  }

  return inner(); // Returns 30
}
```

Closures

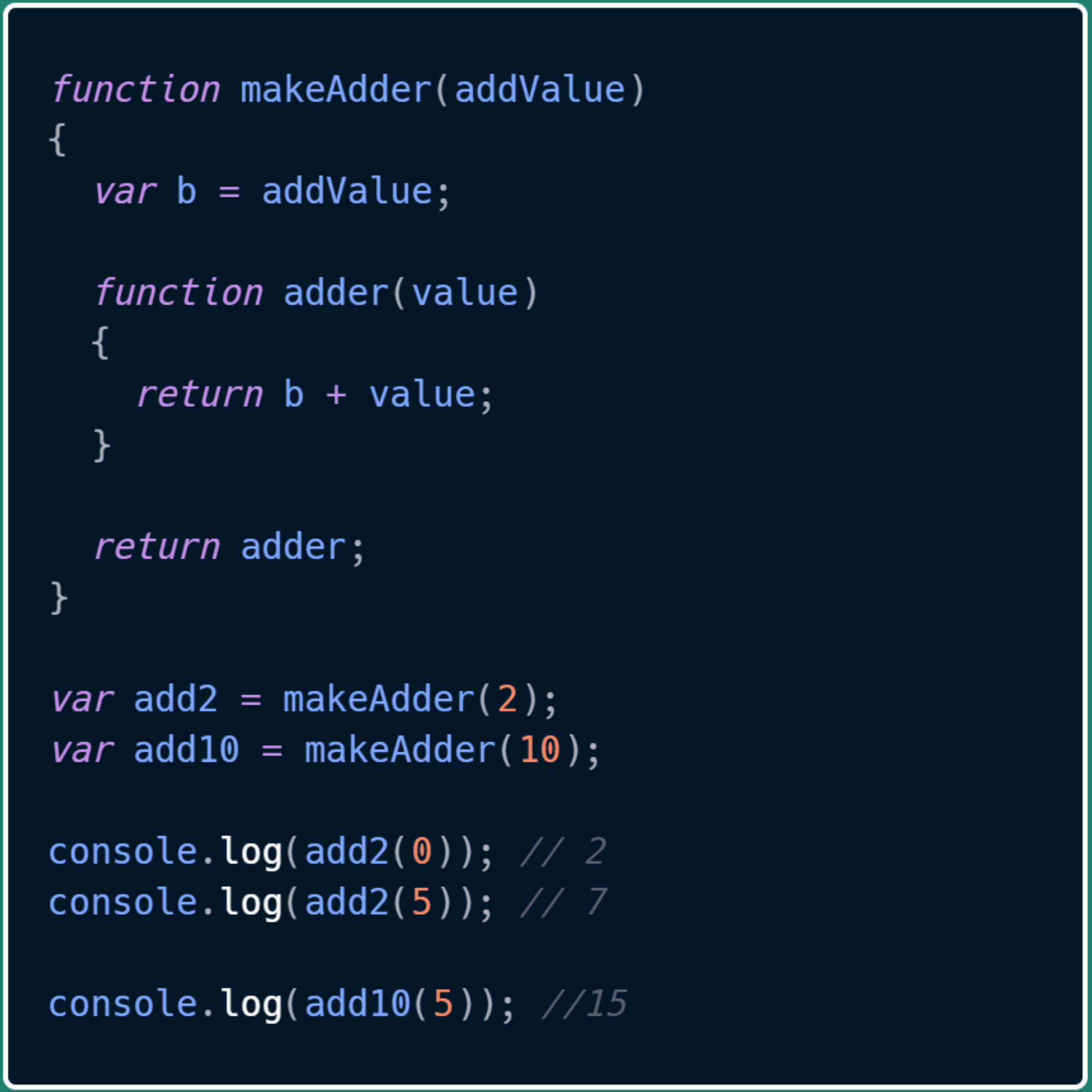
In this case, despite having access only to the inner function, it is possible to retrieve the variable (and its state) of the outer function even if it shouldn't be available after the execution.



```
function outer()
{
  var b = 10;

  function inner()
  {
    var a = 20;

    return a + b;
  }

  return inner; // Returns a function
}

var f = outer();

f(); // Returns 30
```

Closures

With closures it is possible to create “customizable functions” (e.g. filters), simulate objects with a single method but they can also be used to hide implementation detail, i.e. create private variables or functions.



```javascript
function makeAdder(addValue)
{
  var b = addValue;

  function adder(value)
  {
    return b + value;
  }

  return adder;
}

var add2 = makeAdder(2);
var add10 = makeAdder(10);

console.log(add2(0)); // 2
console.log(add2(5)); // 7

console.log(add10(5)); //15
```

CODEPEN

Closure

Arrow functions

An arrow function expression (previously, and now incorrectly known as **fat arrow function**) has a shorter syntax compared to function expressions and does not have its own this.

Arrow functions are always anonymous.

No separate this

Until arrow functions, every new function defined its own *this* value.

This proved to be less than ideal with an object-oriented style of programming.

```javascript
function Person() {
  // The Person() constructor defines `this` as itself.
  this.age = 0;

  setInterval(function growUp() {
    // In nonstrict mode, the growUp() function defines `this`
    // as the global object, which is different from the `this`
    // defined by the Person() constructor.
    this.age++;
  }, 1000);
}

var p = new Person();
```

self / that

```
function Person() {
  var self = this; // Some choose `that` instead of `self`.
                   // Choose one and be consistent.
  self.age = 0;

  setInterval(function growUp() {
    // The callback refers to the `self` variable of which
    // the value is the expected object.
    self.age++;
  }, 1000);
}
```

Problem solved

An arrow function does not have its own *this.*

The *this* value of the enclosing execution context is used.

```
function Person() {
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| properly refers to the person object
  }, 1000);
}

var p = new Person();
```

Javascript Objects

JavaScript is designed on a simple object-based paradigm.

An object is a collection of properties, and a property is an association between a name (or *key*) and a value.

A property's value can be a function, in which case the property is known as a method.

In addition to objects that are predefined in the browser, you can define your own objects.

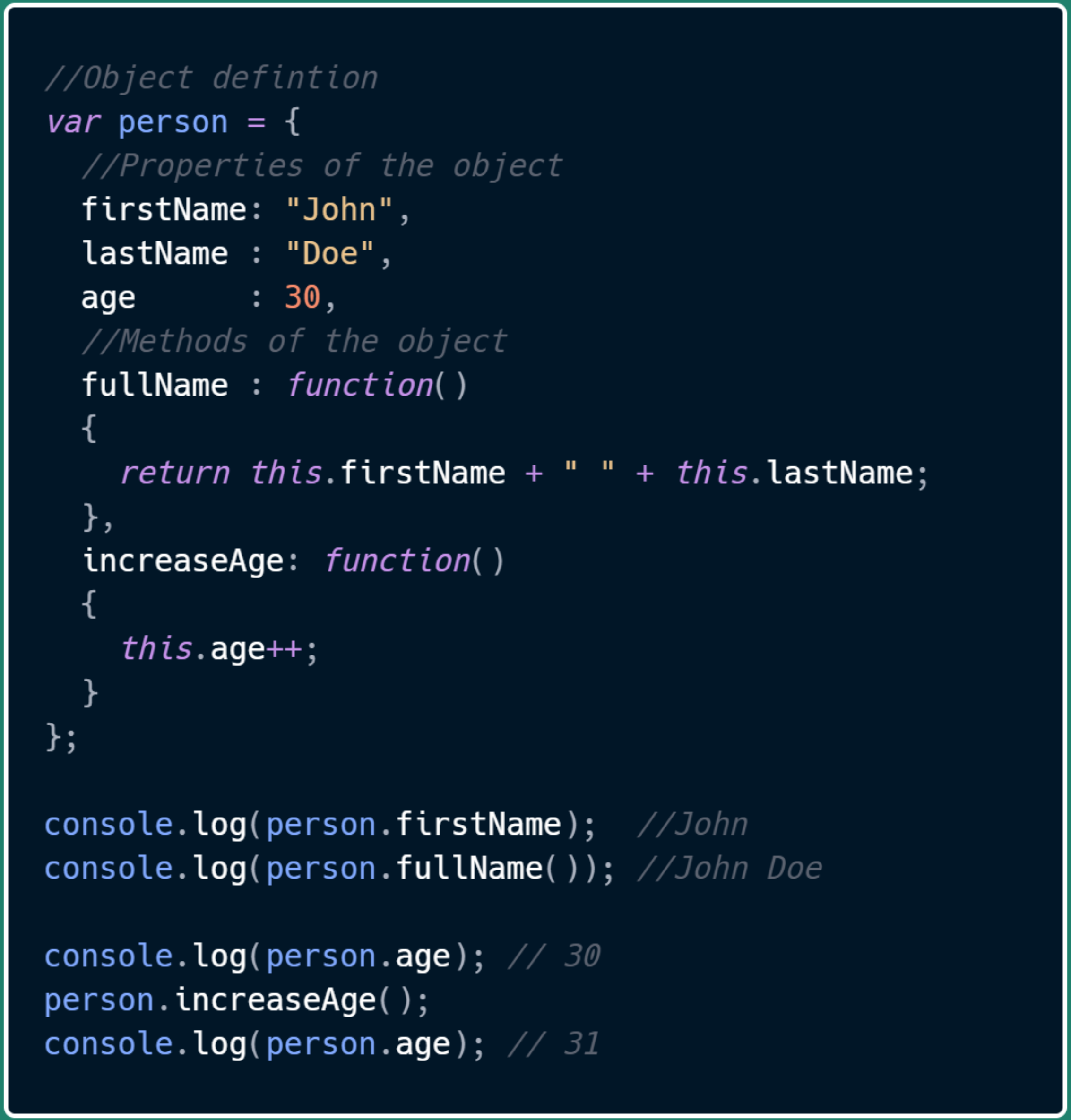
Javascript Objects

The object is defined by enclosing properties and methods between curly braces ({ }).

Each element of the object is composed of expression of ‘key : value/function’ followed by a comma (,) if there are more.

The *this* always refers to the owner of the method.

To access the object’s properties and methods we have to write the name of the variable where the object is stored, followed by a dot (.), followed by the name of the properties/method we need.



```
//Object defintion
var person = {
  //Properties of the object
  firstName: "John",
  lastName : "Doe",
  age      : 30,
  //Methods of the object
  fullName : function()
  {
    return this.firstName + " " + this.lastName;
  },
  increaseAge: function()
  {
    this.age++;
  }
};

console.log(person.firstName);  //John
console.log(person.fullName()); //John Doe

console.log(person.age); // 30
person.increaseAge();
console.log(person.age); // 31
```

Events

Events are actions or occurrences that happen in the system you are programming, which the system tells you about so you can respond to them in some way if desired.

For example, if the user selects a button on a webpage, you might want to respond to that action by displaying an information box.

Web events

There are many different types of events that can occur in your browser.

For example:

- The user selects a certain element or hovers the cursor over a certain element.
- The user chooses a key on the keyboard.
- The user resizes or closes the browser window.
- A web page finishes loading.
- A form is submitted.
- A video is played, paused, or finishes.
- An error occurs.

Complete list of events

Handling events

Each available event has an **event handler**, which is a block of code (written by the programmer) that runs when the event fires.

When such a block of code is defined to run in response to an event, we say we are **registering an event handler**.

Note: Event handlers are sometimes called **event listeners** but strictly speaking, they are different and work together.

The listener listens out for the event happening, and the handler is the code that is run in response to it happening.

CODEPEN

Events

Event handler properties

These are the properties that exist to contain event handler code.

The onclick property is the event handler property being used in this situation.

```
//anonymmous function
const btn = document.querySelector('button');

btn.onclick = function() {
  const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' +
random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}


const btn = document.querySelector('button');

function bgChange() {
  const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' +
random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}

btn.onclick = bgChange;
```

Inline event handlers

```html
<button onclick="bgChange( )">Press me</button>
```

Adding and removing event handlers

The modern mechanism for adding event handlers is the addEventListener() method.

Inside the addEventListener() function, we specify two parameters: the name of the event we want to register this handler for, and the code that comprises the handler function we want to run in response to it.

There is also a counterpart function, removeEventListener()

```
const btn = document.querySelector('button');

// named function
function bgChange() {
  const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' +
random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}
// 1st method: assign the function to the listener
btn.addEventListener('click', bgChange);

// 2nd method: add the event listener and write the function directly inside
the listener
btn.addEventListener('click', function() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255)
+ ')';
  document.body.style.backgroundColor = rndCol;
});

//remove the event listener
btn.removeEventListener('click', bgChange);
```

Event objects

The event handler function receives always a default parameter: specified with a name such as event, evt, or e.

This is called the **event object**, and it is automatically passed to event handlers to provide extra features and information.

The target property of the event object is always a reference to the element the event occurred upon.

CODEPEN

Events

Preventing default behavior

Sometimes, you'll come across a situation where you want to prevent an event from doing what it does by default.

The most common example is that of a web form.

When you submit a login form, the natural behavior is for the data to be submitted to a specified page on the server for processing, and the browser to be redirected to a "success message" page.

The standard Event object has a function available on it called preventDefault() which, when invoked stops the firing of the event.

CODEPEN

[Events]

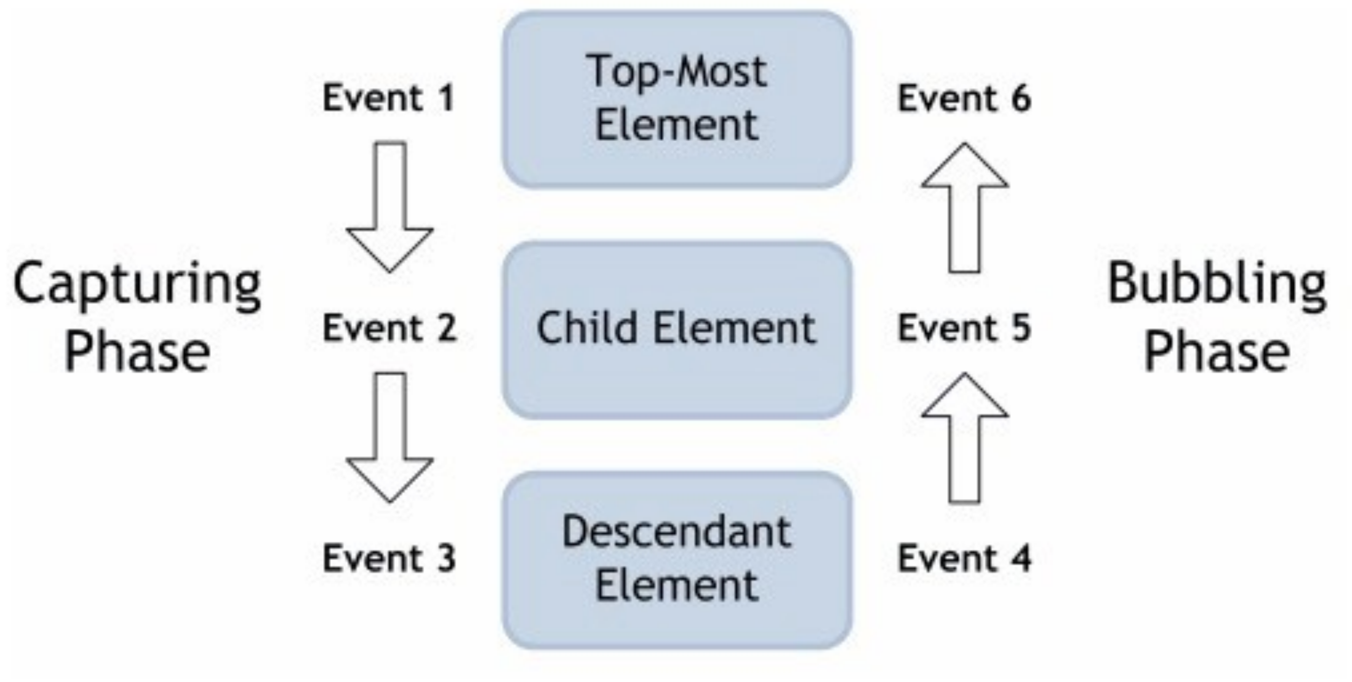
Event bubbling and capture

Event bubbling and capture are two mechanisms that describe what happens when two handlers of the same event type are activated on one element.

In modern browsers, by default, all event handlers are registered for the bubbling phase.

Note: In cases where both types of event handlers are present, bubbling and capturing, the capturing phase will run first, followed by the bubbling phase.

Event bubbling and capture



stopPropagation()

The standard Event object has a function available on it called stopPropagvation() which, when invoked on a handler's event object, makes it so that first handler is run but the event doesn't bubble any further up the chain, so no more handlers will be run.

CODEPEN

stopPropagation