JavaScript– W1S2
Functions and Operators

Cyrille JEGOUREL – Singapore University of Technology and Design

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

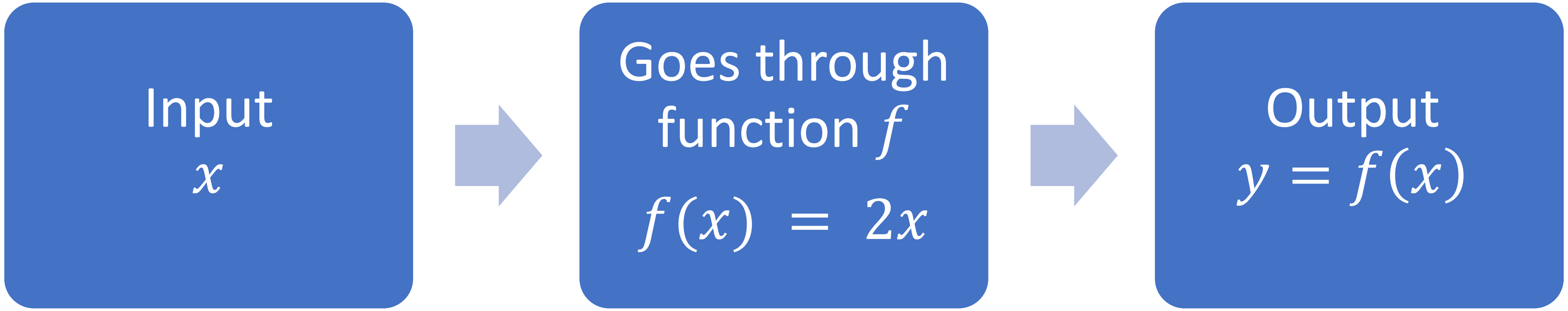
Outline (Week1, Session 2)

- **Functions**:
  - declaration,
  - call,
  - nestedness,
  - global and local scope
- **Operators**:
  - arithmetic,
  - assignment,
  - comparison,
  - logical, etc.

Function objects

- In mathematics, we often like to define **functions**, in the form

$$y = f(x) = 2x$$



Input
x

Goes through function f

$f(x) = 2x$

Output
$y = f(x)$

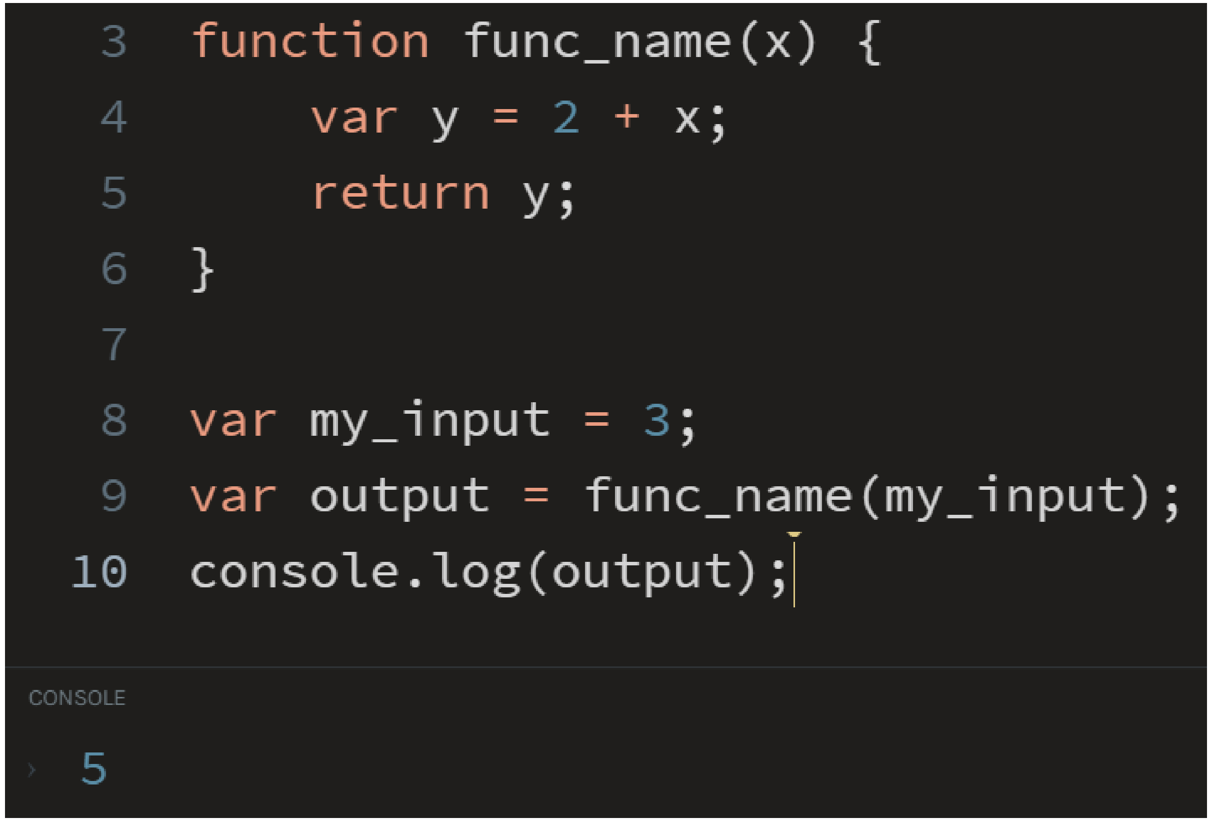
- The same can be done in JavaScript, by creating a function.

Function formatting in JavaScript

- **Definition (JS functions):**
  A JavaScript **function** is a block of code which only runs when it is **called**.

  You can **pass data**, known as **parameters** or **input values**, into a function.

  A function can **return data**, as a **result** or **output values**.

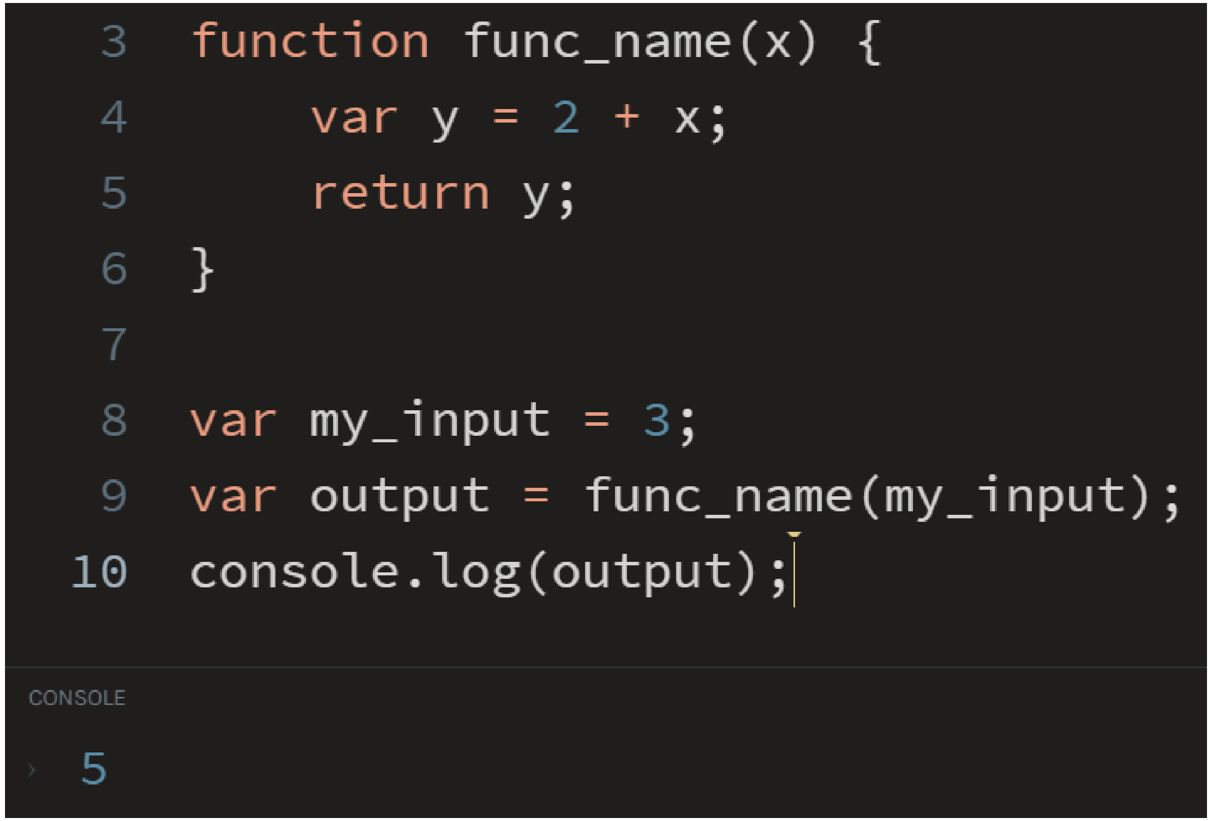


```
function func_name(x) {
    var y = 2 + x;
    return y;
}

var my_input = 3;
var output = func_name(my_input);
console.log(output);
```

```
5
```

Function formatting in JavaScript

- You can define a function, with a **function** statement.
- Immediately after **function**, type the **function's name**.

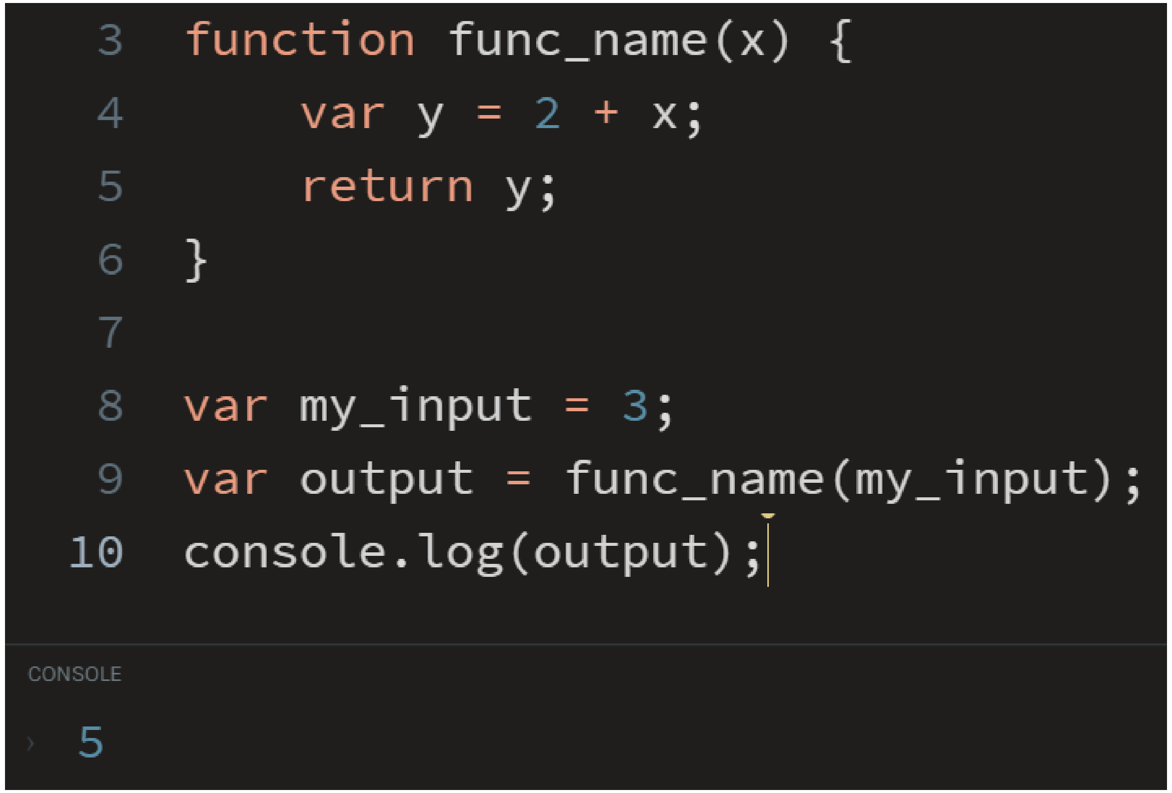


```
function func_name(x) {
    var y = 2 + x;
    return y;
}

var my_input = 3;
var output = func_name(my_input);
console.log(output);
```

```
› 5
```

Function formatting in JavaScript

- You can define a function, with a **function** statement.
- Immediately after **function**, type the **function's name**.
- **Between parentheses**, after the function's name, type **input values/parameters**.
- You don't need to declare the inputs of the function with the **var** keyword.
- The body of the function is then enclosed within curly brackets **{}**.

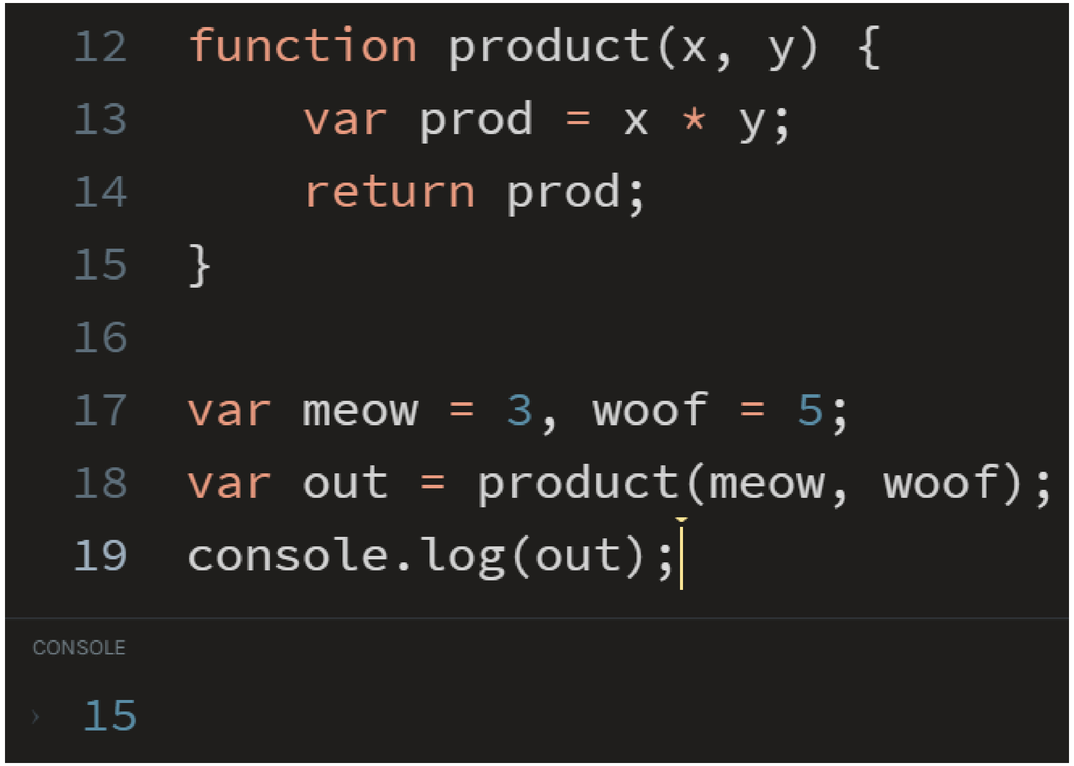


```javascript
function func_name(x) {
    var y = 2 + x;
    return y;
}

var my_input = 3;
var output = func_name(my_input);
console.log(output);
```

```
› 5
```

Function formatting in JavaScript

- **Using the return keyword**, type **output values/results,** that your function should give, if any.
- You may have zero or multiple inputs (and outputs - more on that later) separated with commas.



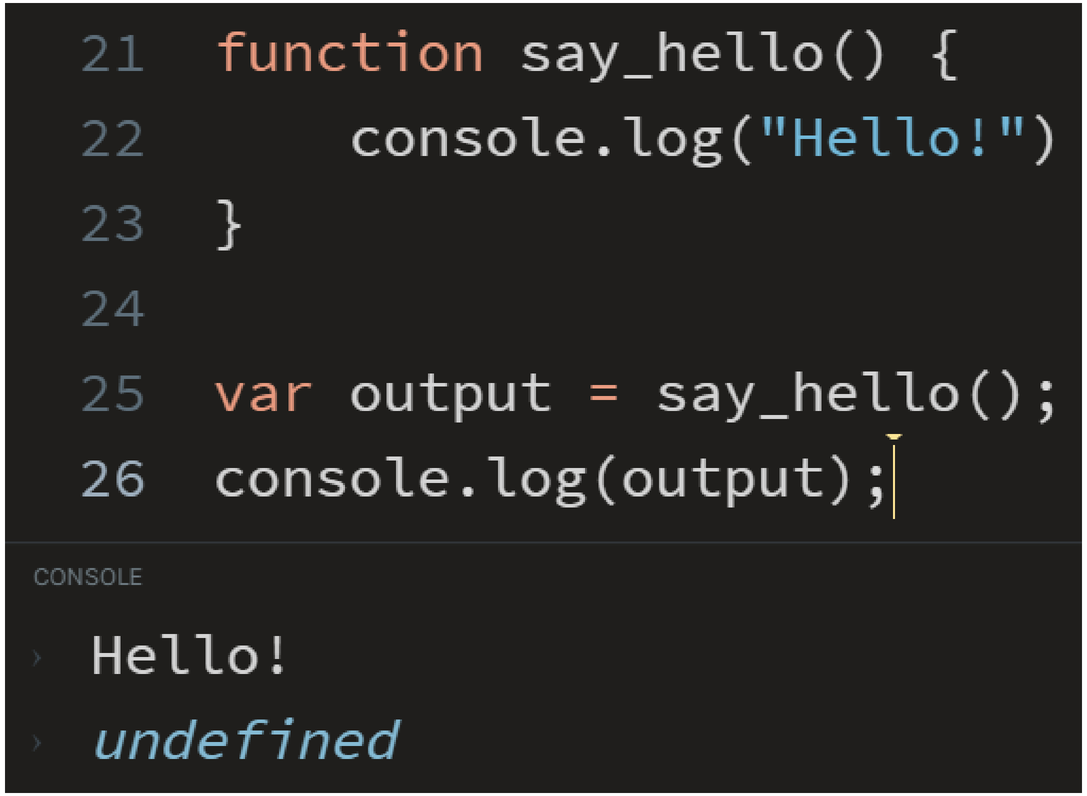
```
function product(x, y) {
    var prod = x * y;
    return prod;
}

var meow = 3, woof = 5;
var out = product(meow, woof);
console.log(out);
```

CONSOLE

```
› 15
```

Function formatting in JavaScript

- **Using the return keyword**, type **output values/results,** that your function should give, if any.
- You may have zero or multiple inputs, (and outputs - more on that later) separated with commas.
- **Note:** Your function may also not **return** anything (that is the case for the **say_hello** function!)



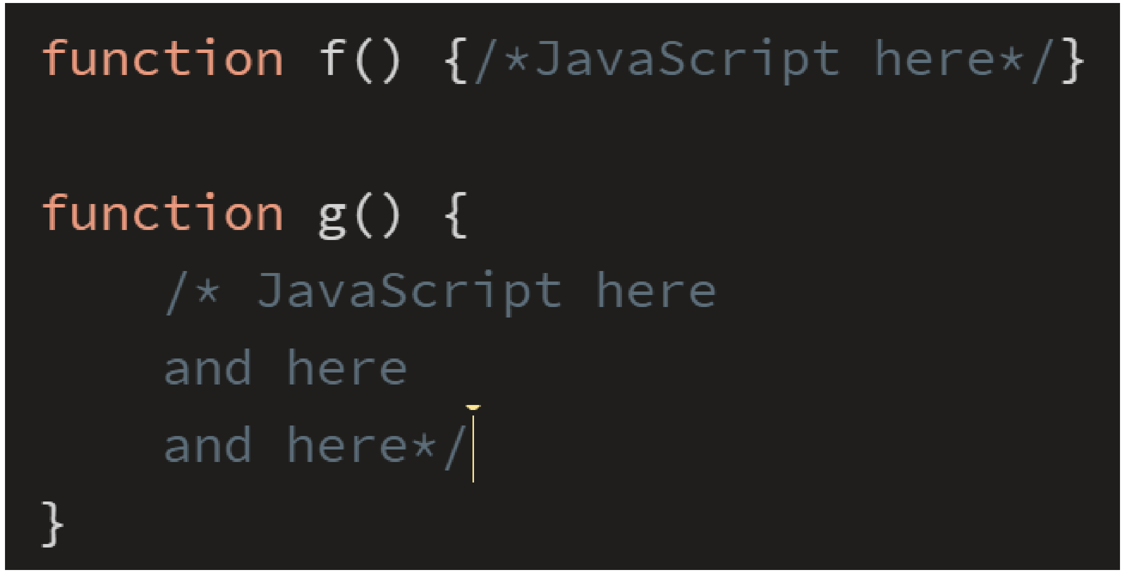
```
function say_hello() {
    console.log("Hello!")
}

var output = say_hello();
console.log(output);
```

CONSOLE

```
Hello!
undefined
```

Function formatting in JavaScript

- **In-between the function and return statement (if any),** you may write lines of code to perform additional / intermediate tasks.
- **Convention** for placing the curly bracket is:
    - to put the opening bracket on the same line of the function declaration.
    - The closing bracket is placed at the same indentation level that **function** statement after the body, unless the function can be fully declared in one line.
    - The body of the function is **indented** with 4 spaces.



```
function f() {/*JavaScript here*/}

function g() {
    /* JavaScript here
    and here
    and here*/
}
```

Great advice #3

Great Advice #3: make your function names explicit.

Same advice than for variable names: it is a good idea to make your functions **explicit**, rather than using meaningless ones that leave your reader guessing (f, g, etc.).

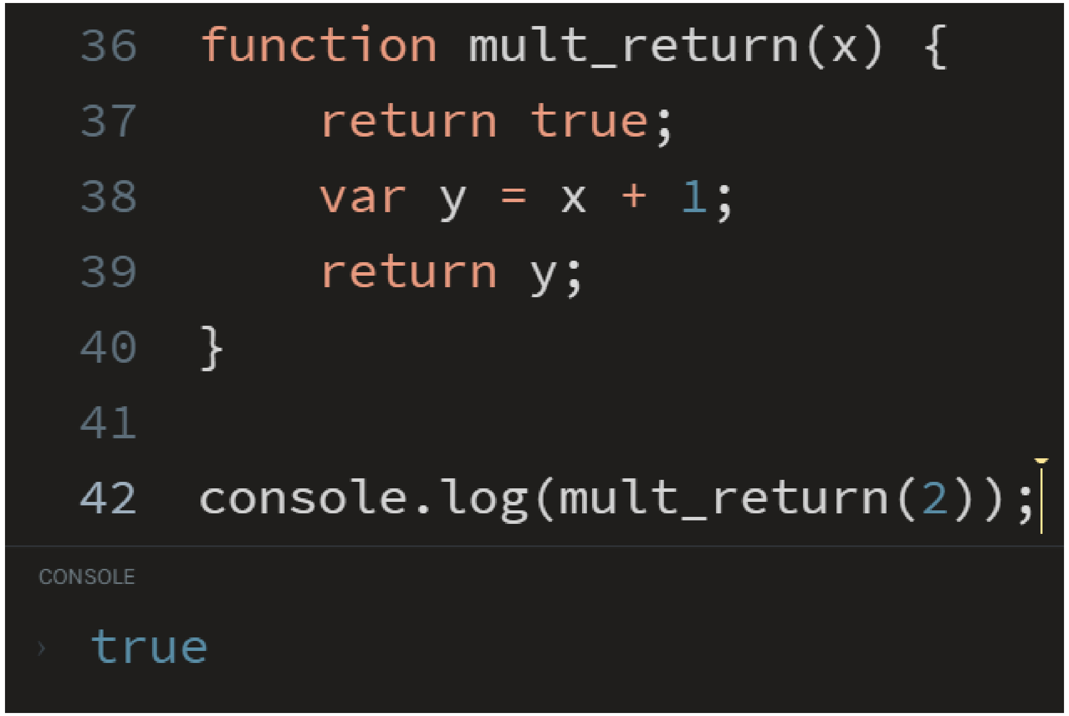
You can **use underscores** if you find it helpful (e.g. function say_hello() {…}).

Your future self, and colleagues, will thank you, when they work on your code later on.

Moreover, same rules than for variable names apply: be aware of case sensitivity, avoid keywords and reserved words, etc.

Function formatting in JavaScript

- **In-between the def and return statement (if any),** you may write lines of code to perform additional/intermediate tasks.
- **Important note:** once a **return** is reached and executed, the function closes and will not execute anything else.
- You can call functions anywhere in your script code, including into other functions (e.g., say_hello() was actually calling console.log() in its body).

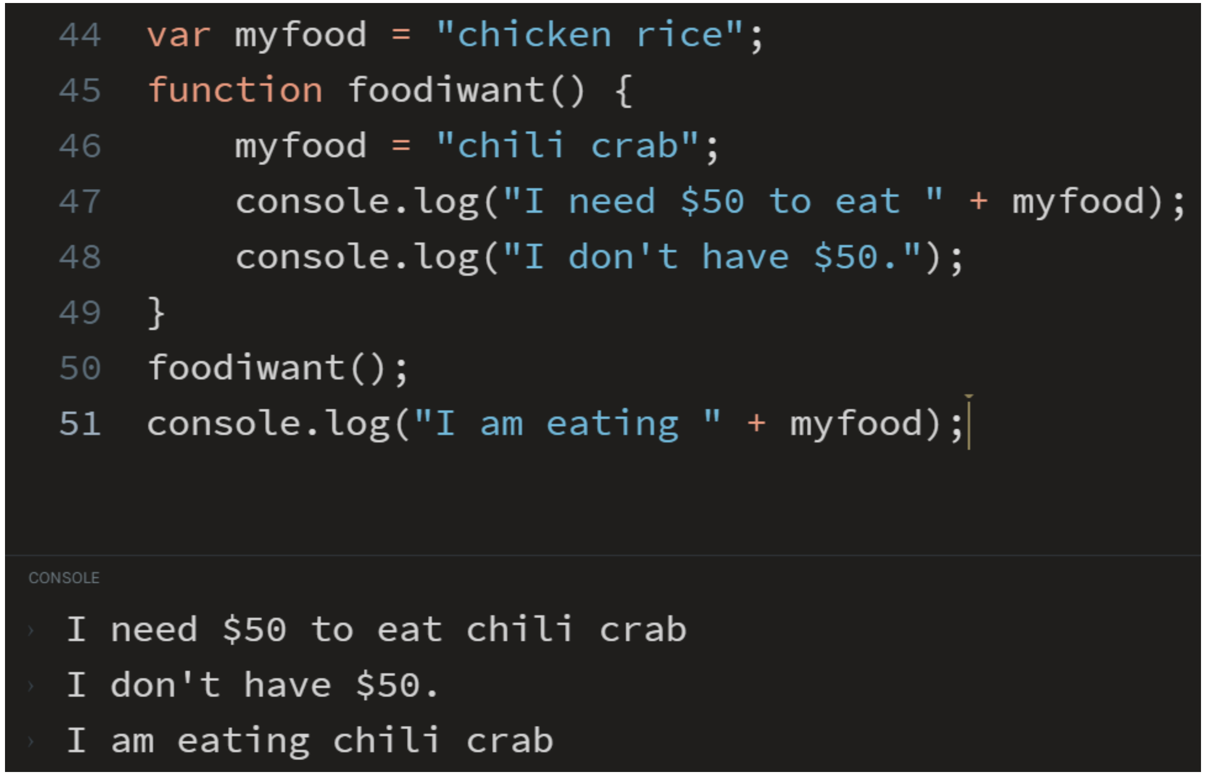


```
function mult_return(x) {
    return true;
    var y = x + 1;
    return y;
}

console.log(mult_return(2));
```

```
› true
```

Global variables

- **Global variables** are variables defined outside of functions and that can be changed anywhere in the script. They are introduced by **var.**
- In the example, myfood is declared as a global variable. In the function, I assigned a new value to myfood.
- Without var, I am not creating a new variable inside the function but changing the value of the global variable.
- Which is not right here. Obviously, I am not eating chili crab at SUTD canteen.

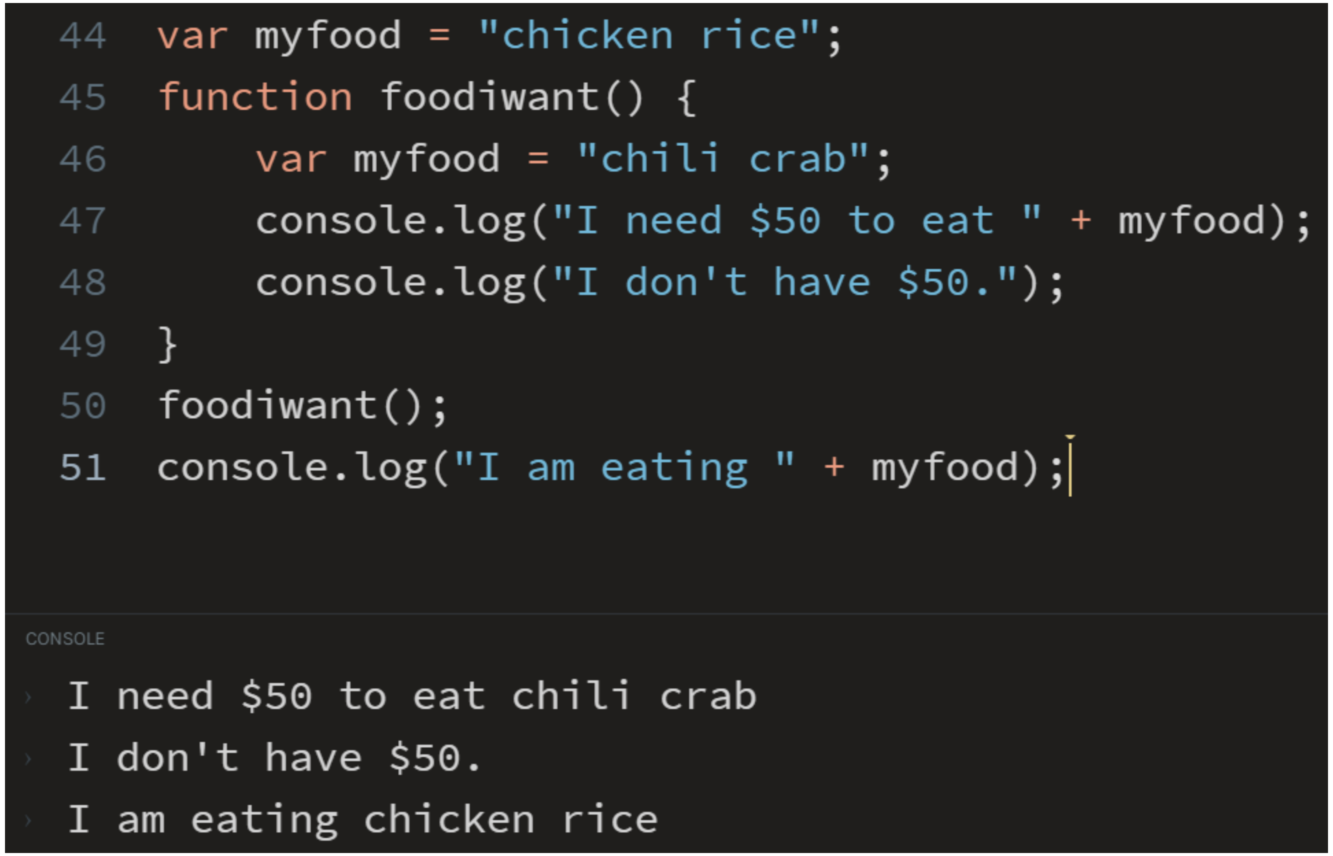


```javascript
var myfood = "chicken rice";
function foodiwant() {
    myfood = "chili crab";
    console.log("I need $50 to eat " + myfood);
    console.log("I don't have $50.");
}
foodiwant();
console.log("I am eating " + myfood);
```

```
I need $50 to eat chili crab
I don't have $50.
I am eating chili crab
```

Local variables

- Which is not right here. Obviously, I am not eating chili crab at SUTD canteen.
- Instead, I should use **local variables**, which can only be used within the function in which it is declared.
- In the example, a local variable myfood is declared in the foodiwant() function. This variable only lives during the execution of the function.
- After the execution of the function however, the global variable myfood is still "alive" and unchanged.

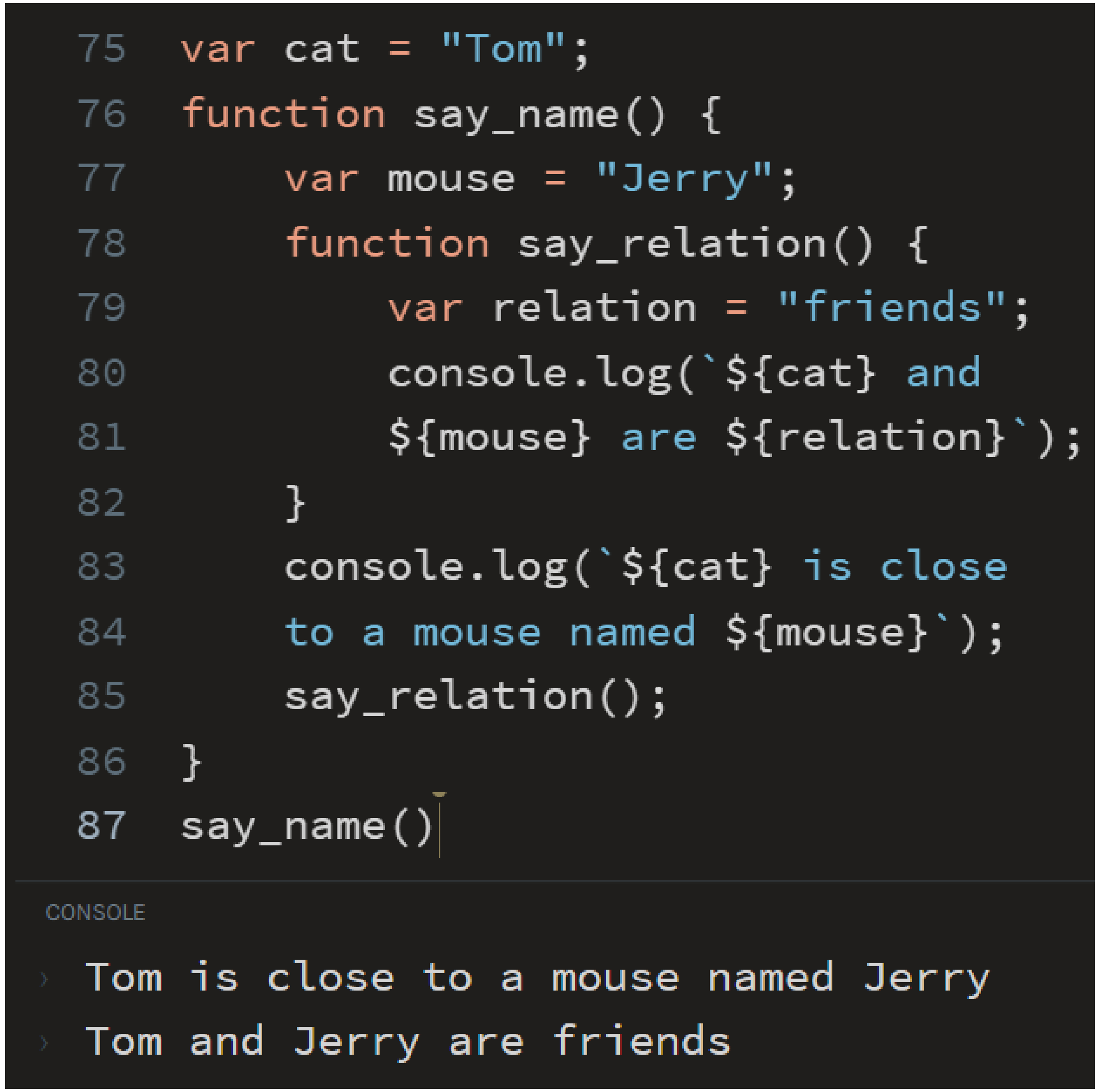


```javascript
var myfood = "chicken rice";
function foodiwant() {
    var myfood = "chili crab";
    console.log("I need $50 to eat " + myfood);
    console.log("I don't have $50.");
}
foodiwant();
console.log("I am eating " + myfood);
```

```
I need $50 to eat chili crab
I don't have $50.
I am eating chicken rice
```

Scope/Context basics (I)

- It is essential to understand the context in which a variable or a function is usable and valid
- **Global context**: includes any code that is not within a function.
- Caution: in a long script, high chance of overwriting accidentally a global variable.
- **Function context**: in JS, a function creates a new context. Any variable declared inside a function with var is not available outside the function. E.g.,
- Global context: cat, say_name()
- say_name() context: cat (from global), mouse, say_relation()
- say_relation() context: cat (from global), mouse (from say_name()), relation.

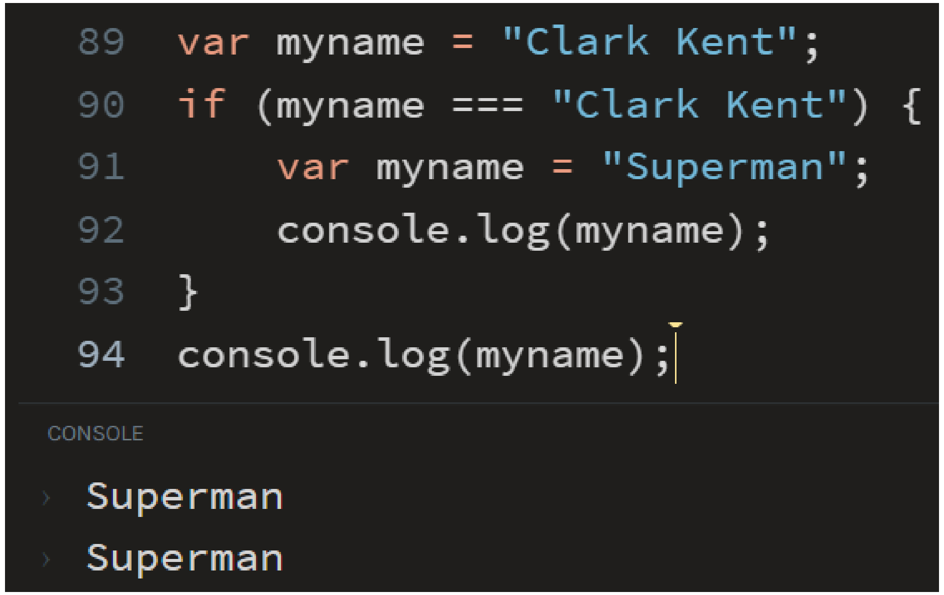


```
var cat = "Tom";
function say_name() {
    var mouse = "Jerry";
    function say_relation() {
        var relation = "friends";
        console.log(`${cat} and
        ${mouse} are ${relation}`);
    }
    console.log(`${cat} is close
    to a mouse named ${mouse}`);
    say_relation();
}
say_name()
```

```
Tom is close to a mouse named Jerry
Tom and Jerry are friends
```

Scope/Context basics (II)

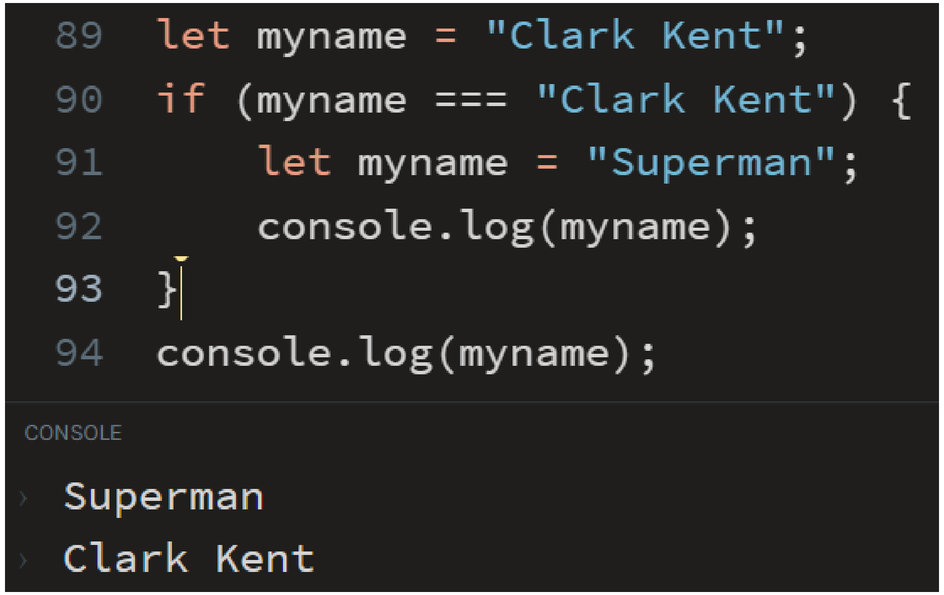
- **Block context**: created for variables defined with the **let** keyword instead of **var**.
- Functions, conditional statement, loops provide a block context between curly brackets (**{}**).
- In a function context, **let** and **var** are similar. However, the advantage come with the possibility to create a block scope for a variable.
- In the 1st example, myname variable is overwritten within the **if** block (more on that later). It is probably an accident.
- But in the 2nd example, myname variable in the **if** block only lives within this block. It does not affect the myname variable(s), global or local, in the other blocks or functions.



```
var myname = "Clark Kent";
if (myname === "Clark Kent") {
    var myname = "Superman";
    console.log(myname);
}
console.log(myname);
```

CONSOLE

```
Superman
Superman
```



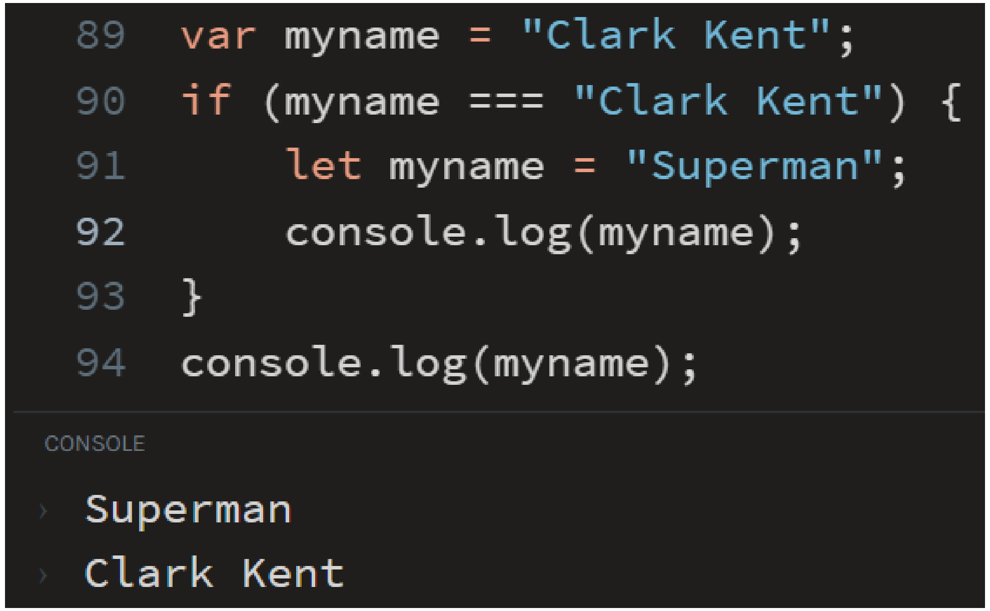
```
let myname = "Clark Kent";
if (myname === "Clark Kent") {
    let myname = "Superman";
    console.log(myname);
}
console.log(myname);
```

CONSOLE

```
Superman
Clark Kent
```

Scope/Context basics (III)

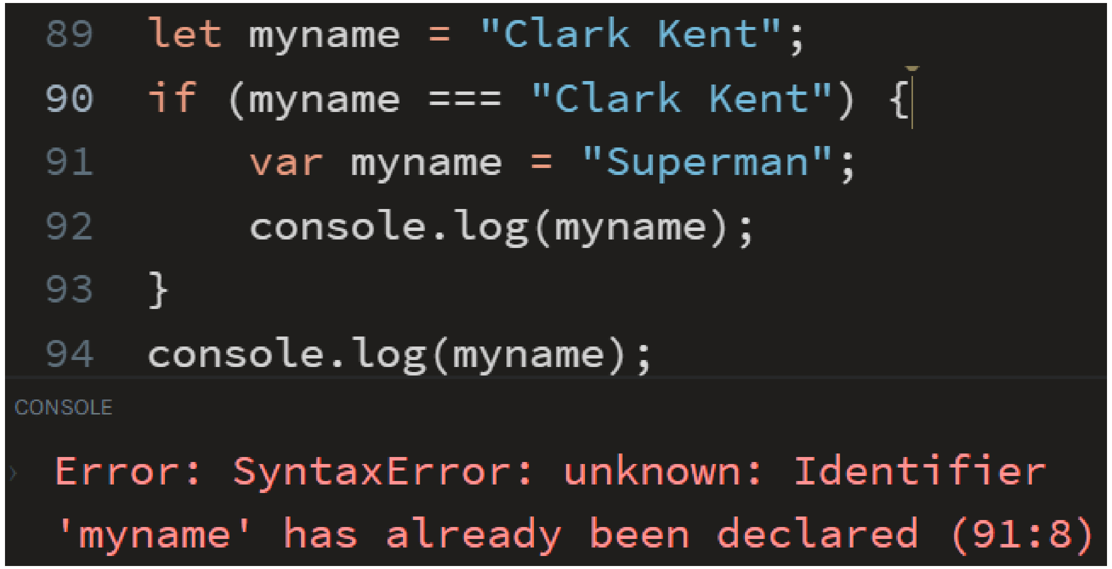
- Same thing here, the myname variable in the **if** block does not affect the global variable with value “Clark Kent”.
- However, you can not declare a global variable with the same name than a variable declared by let in the scope of this let variable.
- Since the scope of a variable is clearer using let, it is recommended to use it instead of var.



```
var myname = "Clark Kent";
if (myname === "Clark Kent") {
    let myname = "Superman";
    console.log(myname);
}
console.log(myname);
```

CONSOLE

```
Superman
Clark Kent
```



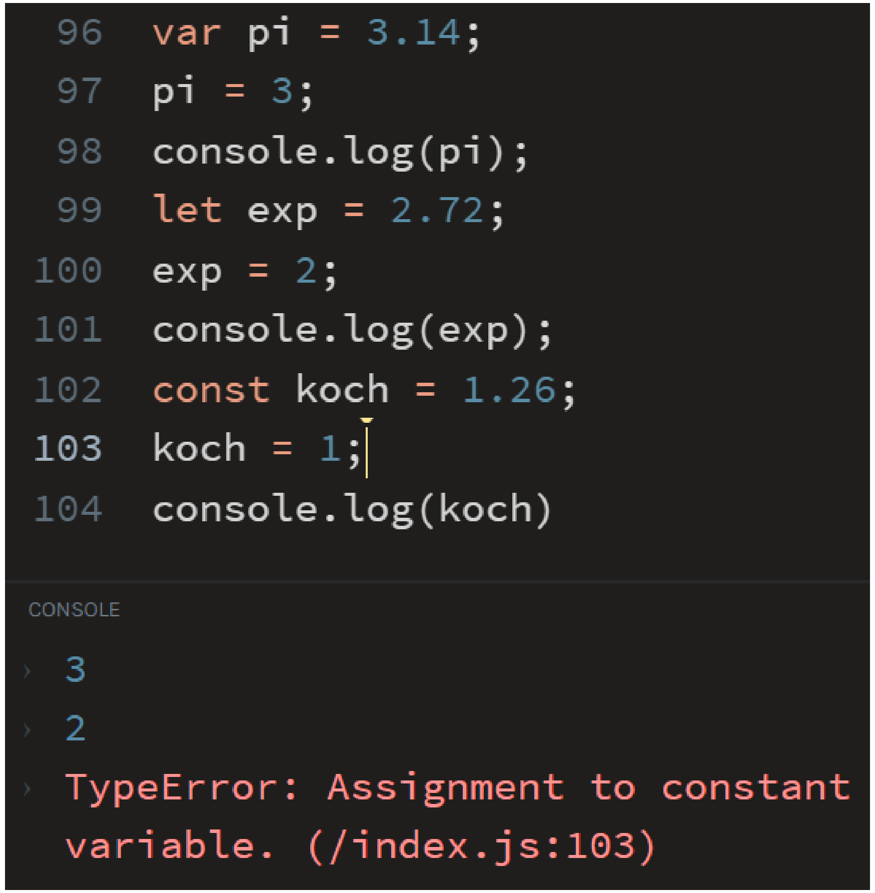
```
let myname = "Clark Kent";
if (myname === "Clark Kent") {
    var myname = "Superman";
    console.log(myname);
}
console.log(myname);
```

CONSOLE

```
Error: SyntaxError: unknown: Identifier
'myname' has already been declared (91:8)
```

Const keyword

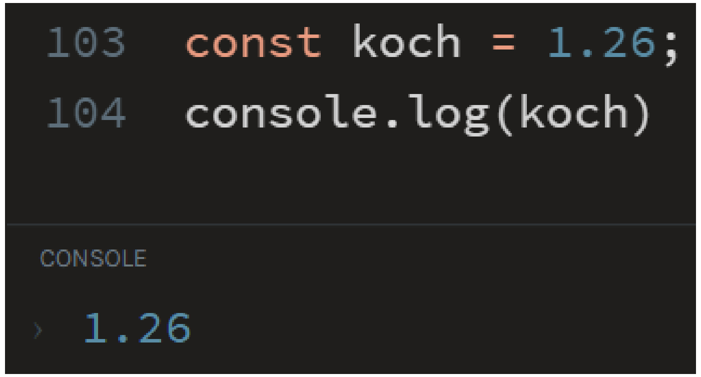
- **const**: used for values that will remain constant.
- If you try to update the value of a variable declared using **const**, the interpreter will throw an error.
- Like let, the **const** keyword uses block scope.
- For this reason, try to use **let** and **const** instead of **var** when defining values, since their scope is clearer.



```
var pi = 3.14;
pi = 3;
console.log(pi);
let exp = 2.72;
exp = 2;
console.log(exp);
const koch = 1.26;
koch = 1;
console.log(koch)
```

CONSOLE

```
3
2
TypeError: Assignment to constant
variable. (/index.js:103)
```



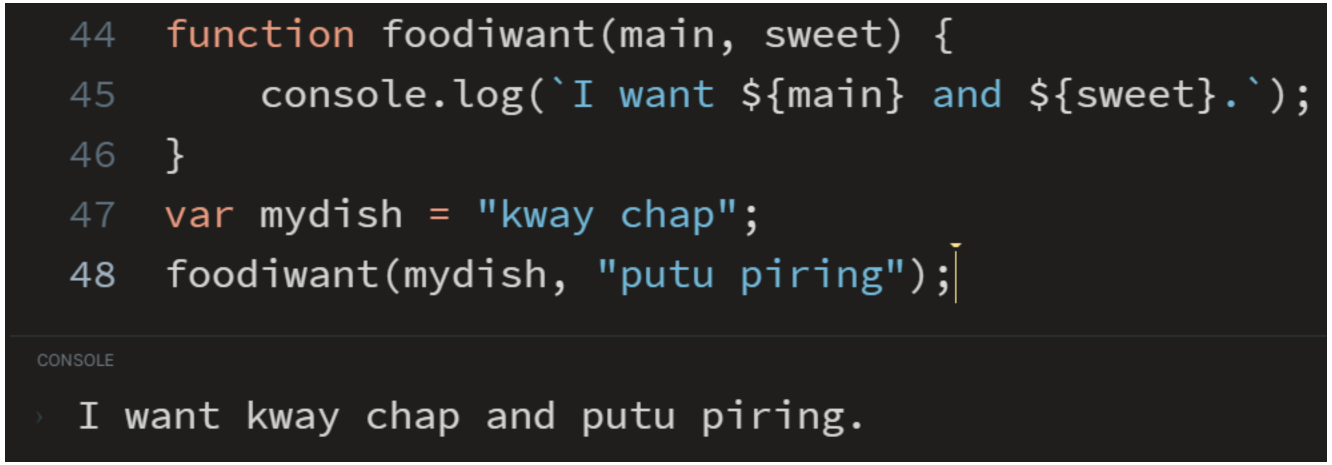
```
const koch = 1.26;
console.log(koch)
```

CONSOLE

```
1.26
```

Using variables and values as function arguments

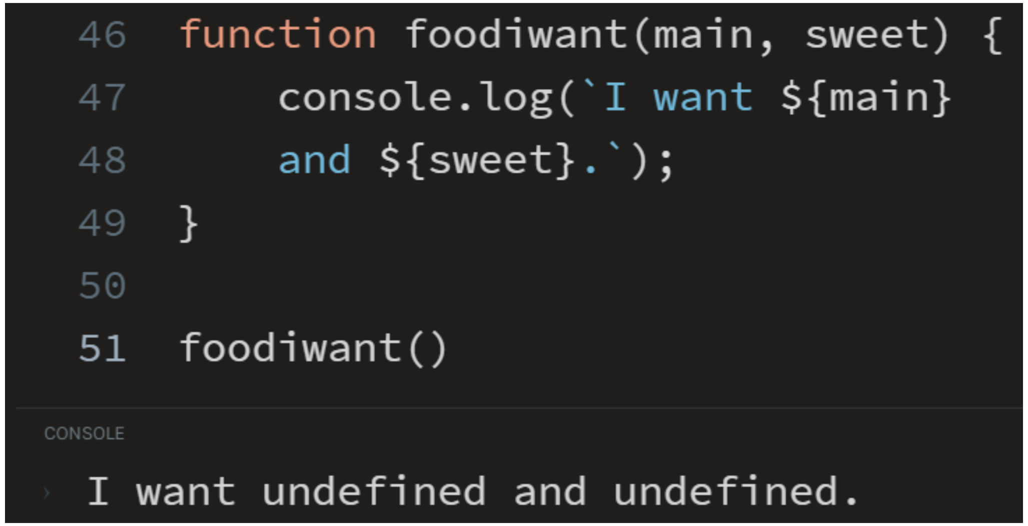
- You can **send a global variable along to a function**. In the example,
- The function takes two arguments, main and sweet, which are local variables inside the function. Note that as argument, they don’t need to be declared with **var**.
- mydish is a **global** variable whom value is assigned to main when the script calls the function.



```
function foodiwant(main, sweet) {
    console.log(`I want ${main} and ${sweet}.`);
}
var mydish = "kway chap";
foodiwant(mydish, "putu piring");
```

```
I want kway chap and putu piring.
```

- You can also **send a value as an argument directly**.
- In the example, “putu piring” is just a value turned into a local variable inside the function.
- Finally, **arguments are optional**. If the function is called without arguments, the local variables main and sweet will take the default value **undefined.**

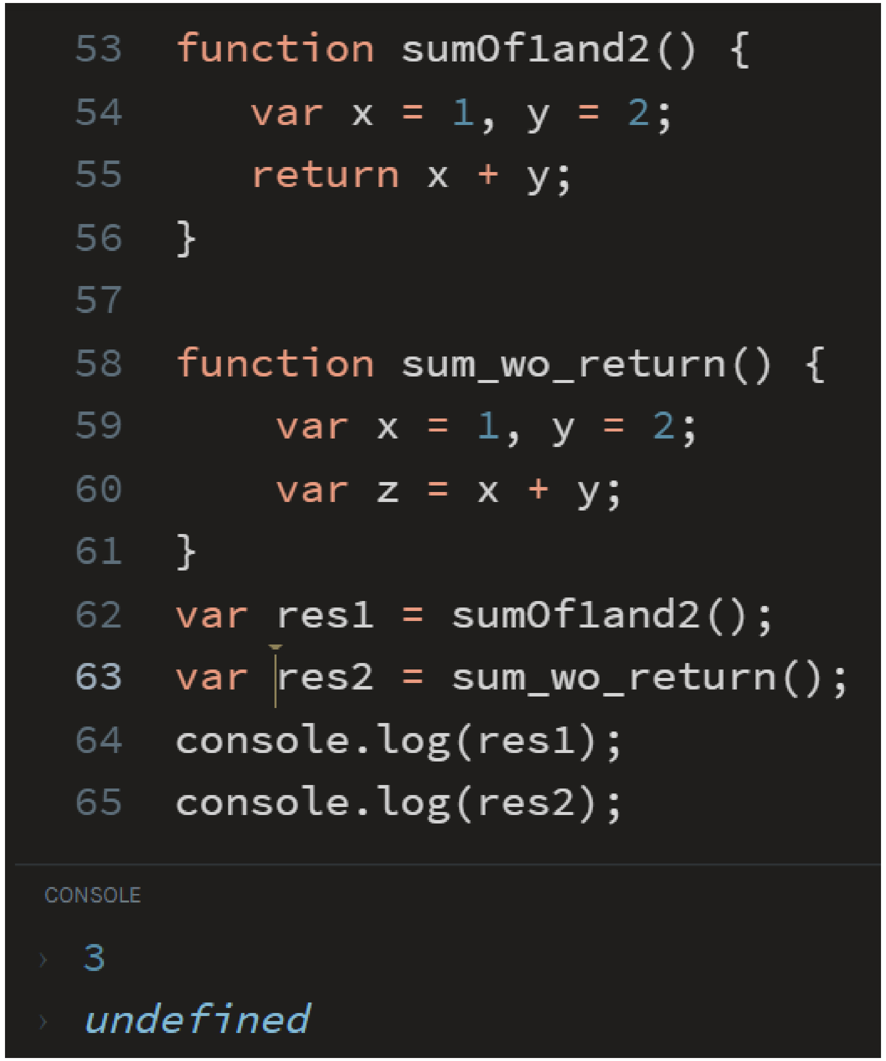


```
function foodiwant(main, sweet) {
    console.log(`I want ${main}
    and ${sweet}.`);
}

foodiwant()
```

```
I want undefined and undefined.
```

Calling functions with return statements

- You can assign the result of a function as the value of a variable.
- Useful because the variable gets the value returned by the function and can be used later in the script.
- By default, the returned value of a function without return is **undefined.**
- So, make sure to clearly identify what should be the arguments and outputs of your functions.



```
function sumOf1and2() {
    var x = 1, y = 2;
    return x + y;
}

function sum_wo_return() {
     var x = 1, y = 2;
     var z = x + y;
}
var res1 = sumOf1and2();
var res2 = sum_wo_return();
console.log(res1);
console.log(res2);
```

```
3
undefined
```

Great advice #4

Great Advice #5: use functions to avoid repetitions of code

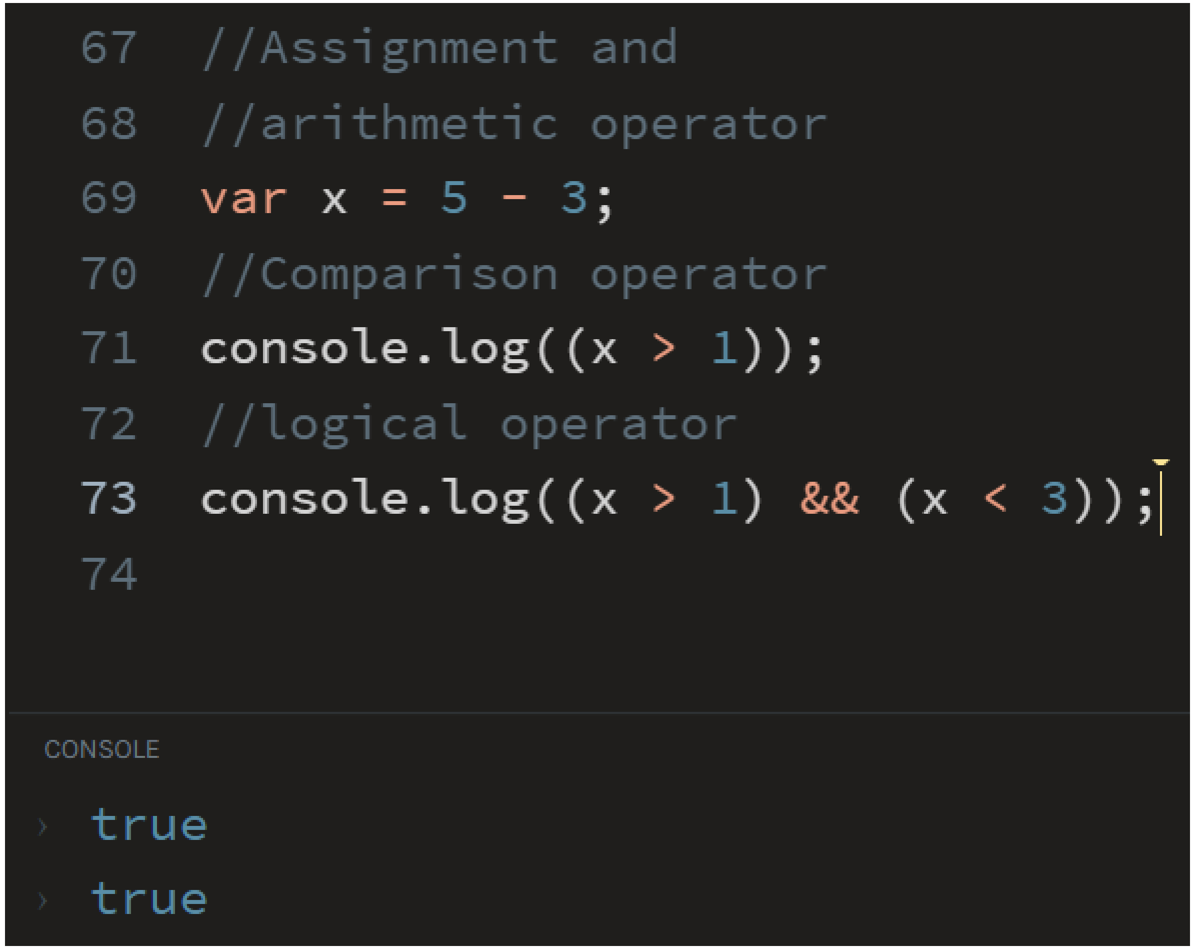
Find yourself copy-pasting blocks of code, and only changing a few values in this block of code?

Then, you probably need a function of some sort, which is called multiple times.

Functions makes your coding easier, and you code look a lot more modular and professional.

Operators

- Operator: JS symbol or keyword that performs some calculation, comparison, assignment on values.
- Different types of operators:
  - **Arithmetic**: perform mathematical calculations on two values (e.g., +, -, *, etc.)
  - **Assignment**: assign values to variables (e.g., =)
  - **Comparison**: compare two values (e.g., “is greater than” (>))
  - **Logical**: compare two conditional statements (e.g., and (&&), or (||))
  - **Bitwise**: logical operators that work at the bit level (out of scope).
  - **Special**: perform other special functions on their own.



```
//Assignment and
//arithmetic operator
var x = 5 - 3;
//Comparison operator
console.log((x > 1));
//logical operator
console.log((x > 1) && (x < 3));
```

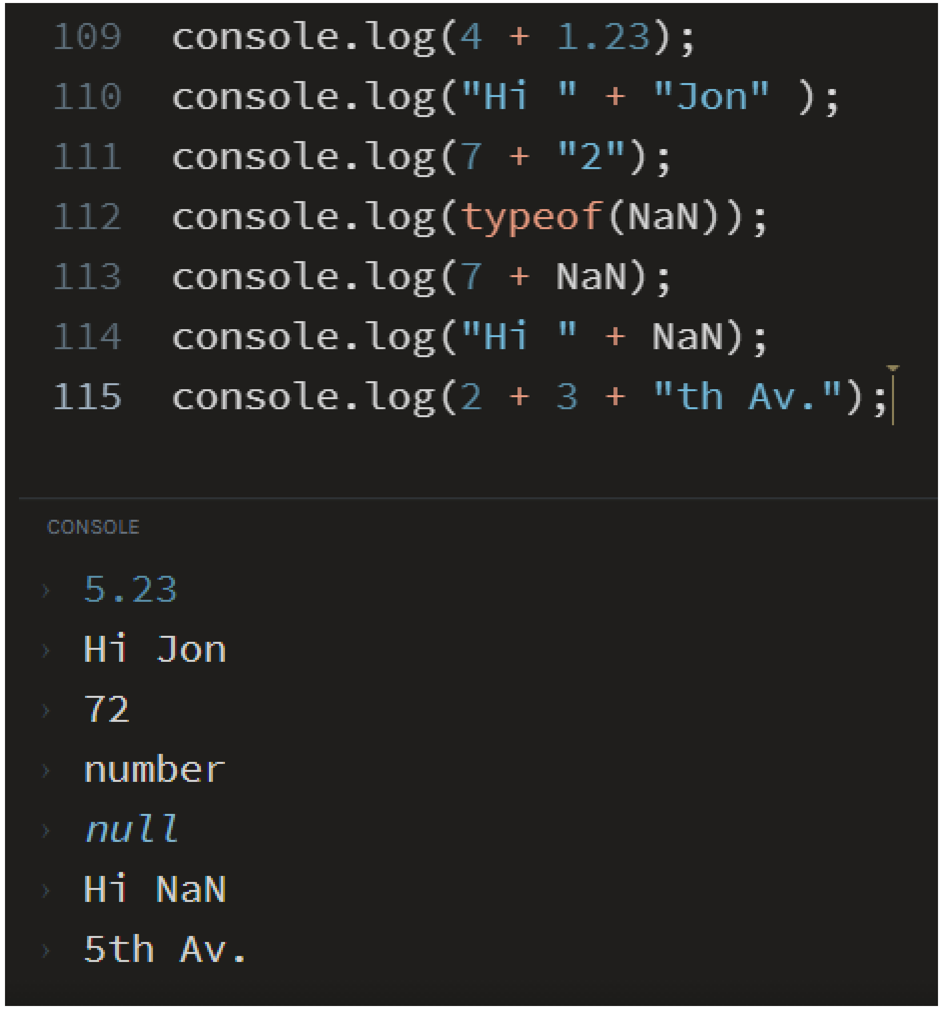
```
true
true
```

Arithmetic Operators

Operator	Symbol	Function	Example
Addition	+	Adds two values	7 + 2 returns 9
Subtraction	-	Subtracts one value from another	7 – 2 returns 5
Multiplication	*	Multiplies two values	7 * 2 returns 14
Division	/	Divides one value by another	7 / 2 returns 3.5
Modulus	%	Returns the remainder	7 % 2 returns 1
Increment	++	Shortcut to add 1 to a number	7++ returns 8
Decrement	--	Shortcut to subtract 1 to a number	7-- returns 6
Unary plus	+	Leaves numeric values as is but will attempt to change non-numeric values into numbers	typeof(+”7”) returns number
Unary negation	-	Reverses the sign of a number	-7 < 0 returns true
Exponentiation	**	Power of a number by another	7 ** 2 returns 49

The Addition operator (+)

- Performs addition between numbers and concatenation between strings.
- No distinction between floats and integers in JS.
- Type coercion: JS attempts to change the data type of a value if necessary. E.g., adding a number to a string will first convert the number into a string and will then concatenate both strings.
- Adding a number and NaN returns NaN or null (depending on your interpreter / version).
- Additions are evaluated from left to right.

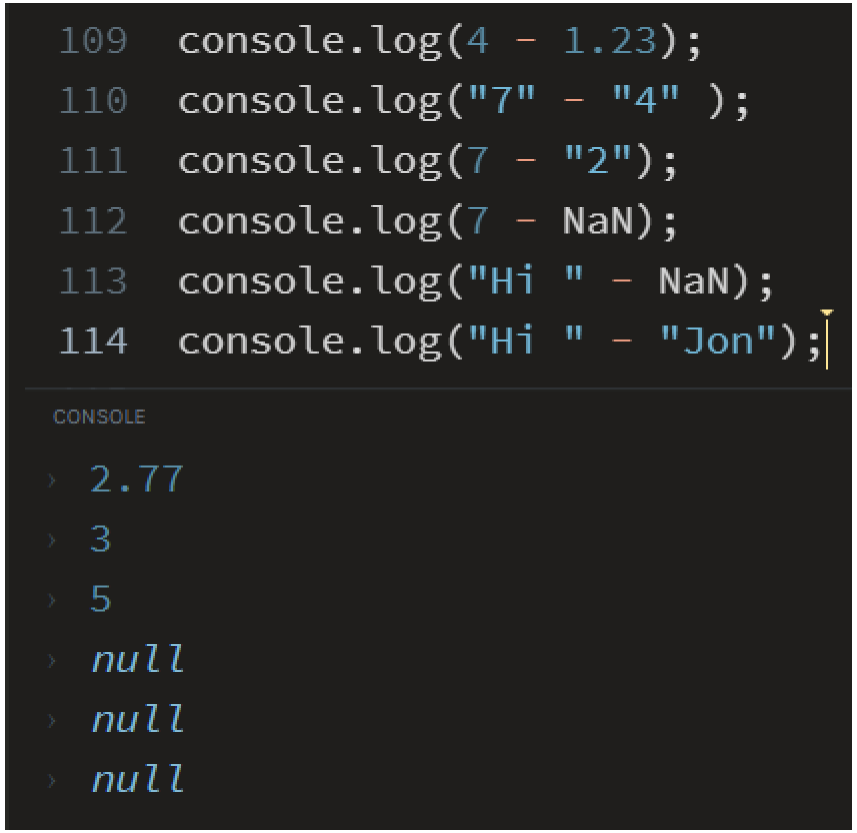


```
console.log(4 + 1.23);
console.log("Hi " + "Jon" );
console.log(7 + "2");
console.log(typeof(NaN));
console.log(7 + NaN);
console.log("Hi " + NaN);
console.log(2 + 3 + "th Av.");
```

```
5.23
Hi Jon
72
number
null
Hi NaN
5th Av.
```

The Subtraction operator (-)

- Performs subtraction between numbers.
- No distinction between floats and integers in JS.
- Type coercion: JS attempts to change the data type of a value if necessary. E.g., subtracting a digit string from another digit string will first convert both strings into numbers and will then evaluate the subtraction.
- If type coercion does not work, the result is in general a null or a NaN.
- If one of the operands in NaN, the result is null or NaN.
- Subtractions are evaluated from left to right.



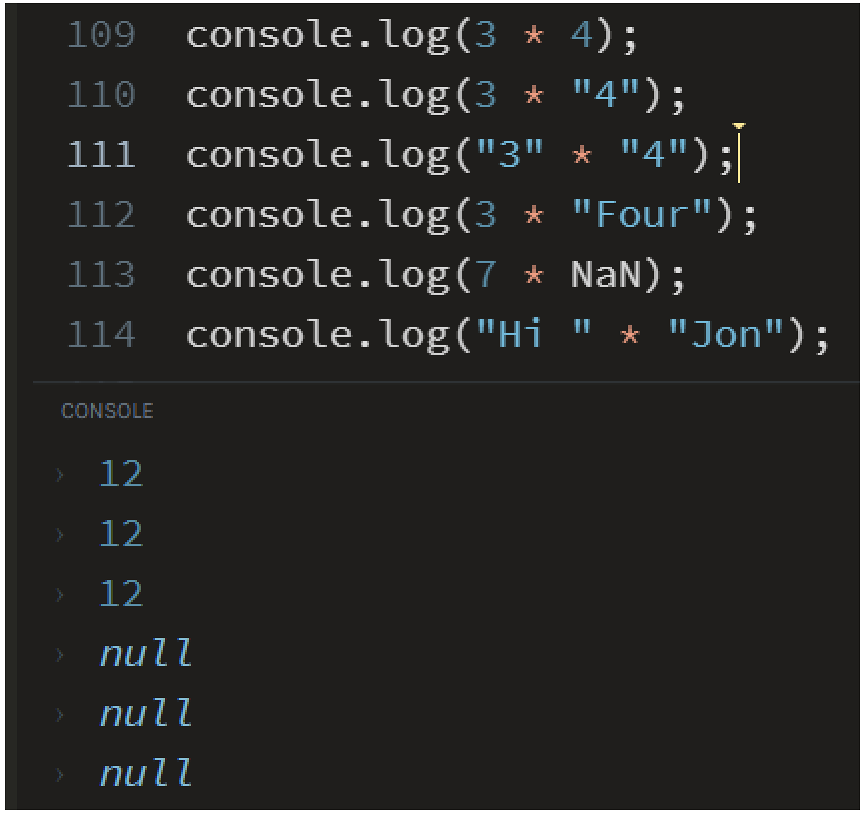
```
console.log(4 - 1.23);
console.log("7" - "4" );
console.log(7 - "2");
console.log(7 - NaN);
console.log("Hi " - NaN);
console.log("Hi " - "Jon");
```

CONSOLE

```
› 2.77
› 3
› 5
› null
› null
› null
```

The Multiplication operator (*)

- Performs multiplication between numbers.
- No distinction between floats and integers in JS.
- Type coercion: JS attempts to change the data type of a value if necessary. E.g., multiplying a digit string with another digit string will first convert both strings into numbers and will then evaluate the multiplication.
- If type coercion does not work, the result is in general a null or a NaN.
- If one of the operands in NaN, the result is null or NaN.
- Multiplications are evaluated from left to right.

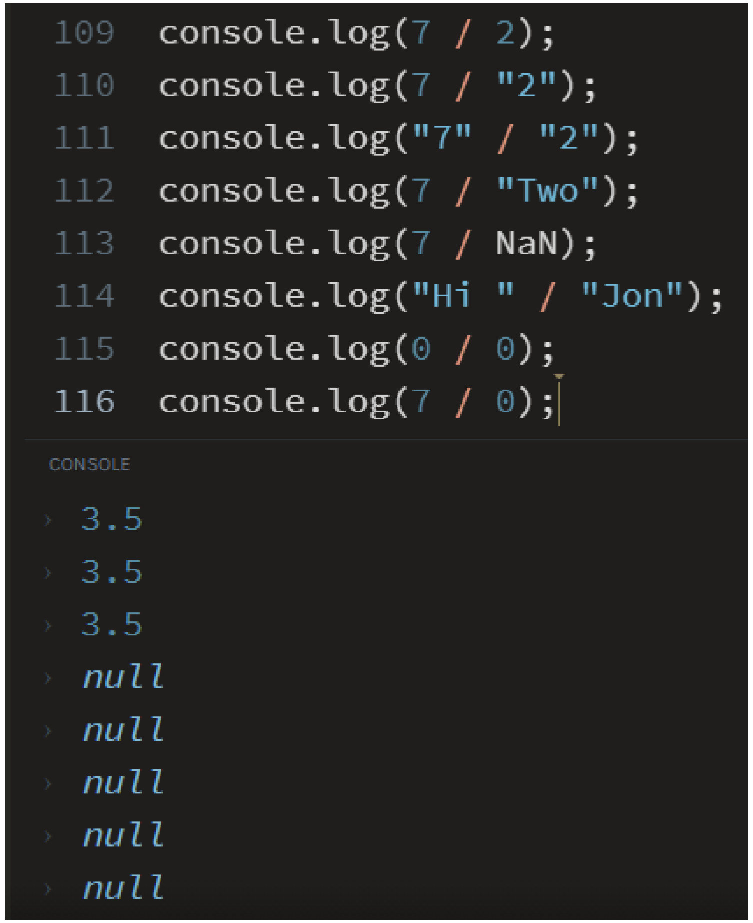


```
console.log(3 * 4);
console.log(3 * "4");
console.log("3" * "4");
console.log(3 * "Four");
console.log(7 * NaN);
console.log("Hi " * "Jon");
```

```
12
12
12
null
null
null
```

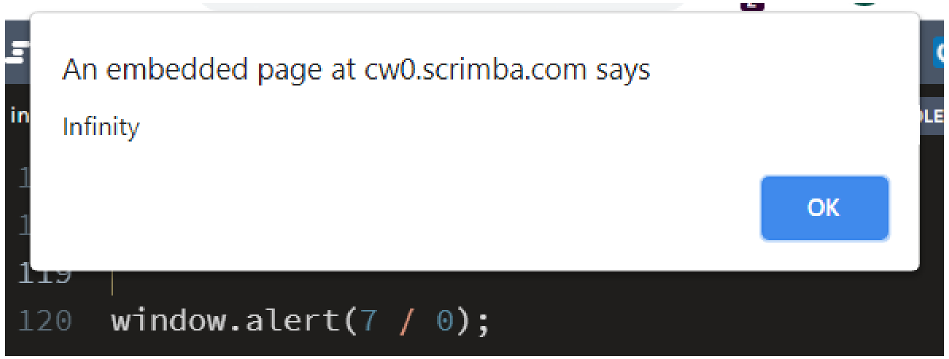
The Division operator (/)

- Performs standard division between numbers.
- No integer division in JS.
- Type coercion: JS attempts to change the data type of a value if necessary. E.g., dividing a digit string with another digit string will first convert both strings into numbers and will then evaluate the division.
- If type coercion does not work, the result is in general a null or a NaN.
- If one of the operands in NaN, the result is null or NaN.
- Divisions are evaluated from left to right.
- Caution with divisions by 0, the result might be null, NaN or even Infinity in some alert box.



```
console.log(7 / 2);
console.log(7 / "2");
console.log("7" / "2");
console.log(7 / "Two");
console.log(7 / NaN);
console.log("Hi " / "Jon");
console.log(0 / 0);
console.log(7 / 0);
```

```
3.5
3.5
3.5
null
null
null
null
null
```



An embedded page at cw0.scrimba.com says

Infinity

```
window.alert(7 / 0);
```

Activity 1: compute the roots of a quadratic equation, with strictly positive determinant

- Consider the quadratic equation

$$ax^2 + bx + c = 0$$

- With $a, b, c \in \mathbb{R}$, such that the determinant $\Delta > 0$

$$\Delta = b^2 - 4ac$$

- When they exist, the two roots of this equation (x_1, x_2) are given by

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \quad and \quad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

Activity 1: compute the roots of a quadratic equation, with strictly positive determinant

- Consider the quadratic equation
$$ax^2 + bx + c = 0$$
- With $a, b, c \in \mathbb{R}$, such that the determinant $\Delta > 0$
$$\Delta = b^2 - 4ac$$
- When they exist, the two roots of this equation (x_1, x_2) are given by

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \quad and \quad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

- **Task:** Consider a quadratic equation with arbitrary $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ coefficients.

Write and test a JS function

1. that **computes** the value of Δ and **prints** it on screen (to check it is strictly positive)
2. And, later on, **computes** the values of the roots x_1 and x_2 and **displays** them on screen.

Activity 1: compute the roots of a quadratic equation, with strictly positive determinant

- For this activity, you may output several solutions under the form of an empty array [] if no solution or an array with 2 elements (which can be sometimes equal) [solution_1, solution_2].
- Also, remember, the **square root of $\boldsymbol{x}$** is equivalent to $\boldsymbol{x}$ **being exponentiated to the power $\mathbf{0.5}$**

$$\sqrt{x} = x^{0.5} = x ** (0.5)$$

Activity 1: compute the roots of a quadratic equation, with strictly positive determinant

- Consider the quadratic equation

$$ax^2 + bx + c = 0$$

- With $a, b, c \in \mathbb{R}$, such that the determinant $\Delta > 0$

$$\Delta = b^2 - 4ac$$

- When they exist, the two roots of this equation (x_1, x_2) are given by

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \quad and \quad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

- **Task:** Consider a quadratic equation with arbitrary $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ coefficients.

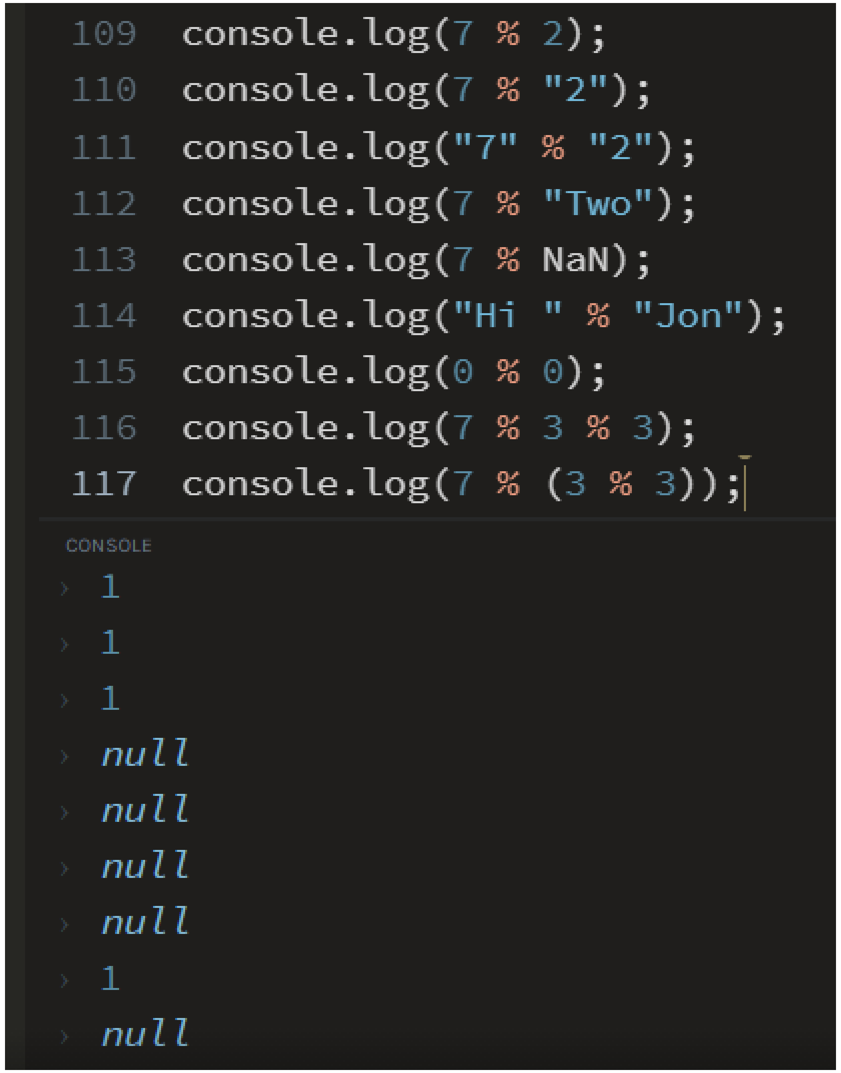
Write and test a JS function

1. that **computes** the value of Δ and **prints** it on screen (to check it is strictly positive)
2. And, later on, **computes** the values of the roots x_1 and x_2 and **displays** them on screen.

→ **Expected answers with a = 2, b = $-$2, c = $-$24: $\boldsymbol{\Delta = 196, x_1 = 4, x_2 = -3}$.**

The Modulus operator (%)

- Evaluates the remainder of the integer division of a number by another.
- Type coercion: As usual, JS attempts to change the data type of a value if necessary.
- If type coercion does not work, the result is in general a null or a NaN.
- If one of the operands in NaN or if the right operand is zero, the result is null or NaN.
- Modulus are evaluated from left to right.



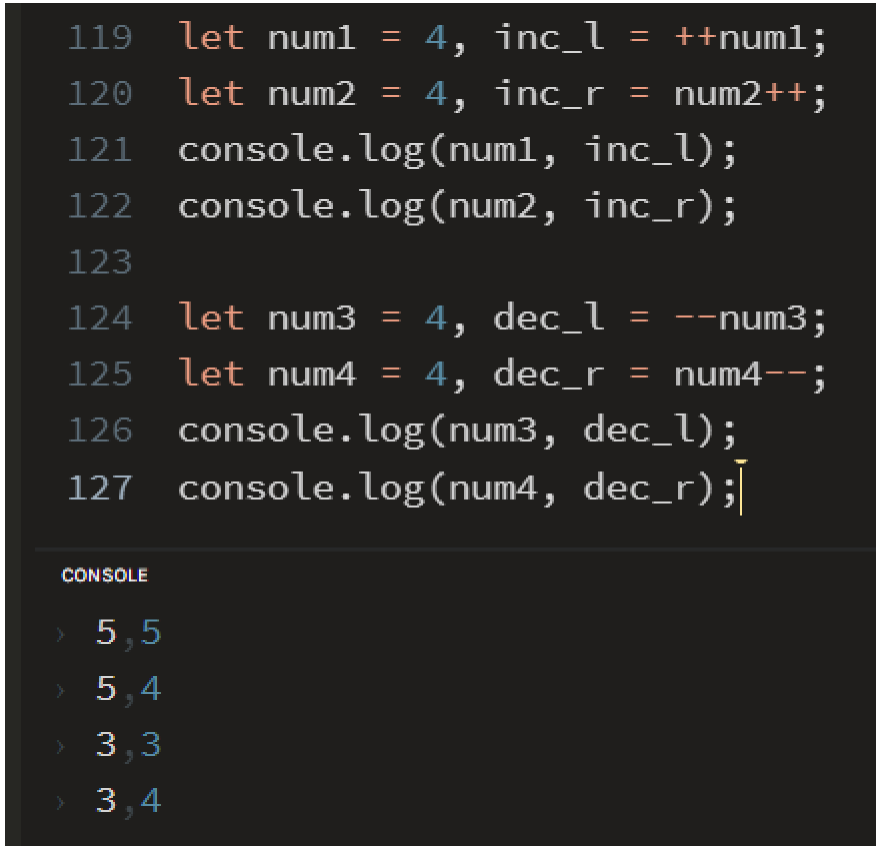
```
console.log(7 % 2);
console.log(7 % "2");
console.log("7" % "2");
console.log(7 % "Two");
console.log(7 % NaN);
console.log("Hi " % "Jon");
console.log(0 % 0);
console.log(7 % 3 % 3);
console.log(7 % (3 % 3));
```

CONSOLE

```
› 1
› 1
› 1
› null
› null
› null
› null
› 1
› null
```

The Increment and decrement operators (++, --)

- The increment and decrement operators are unary and respectively increase or decrease the value of its lone operand by 1.
- When the increment/decrement operator is placed before the operand, it modifies the operand by +/- 1, and then the rest of the statement is executed.
- When the increment/decrement operator is placed after the operand, it changes the value of the operand after the assignment.
- E.g., num1 and num3 begins with the value of 4. However, when the code assigns the values to variables inc_l and dec_l, it first updates num1 and num3 and then assign the results: (5,5 and 3,3)
- But inc_r and dec_r will be assigned the current values of num2 and num4 before updating num2 and num4: (5,4 and 3,4)

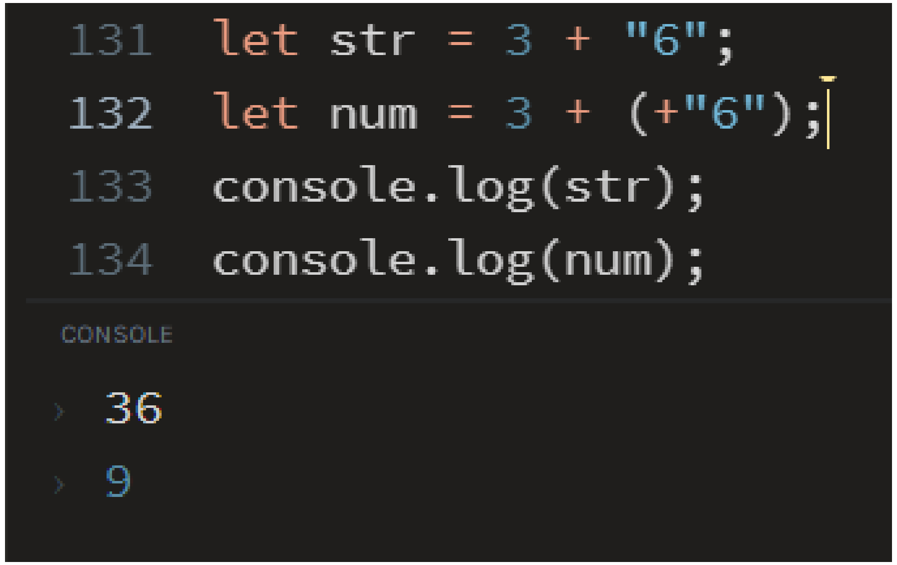


```
let num1 = 4, inc_l = ++num1;
let num2 = 4, inc_r = num2++;
console.log(num1, inc_l);
console.log(num2, inc_r);

let num3 = 4, dec_l = --num3;
let num4 = 4, dec_r = num4--;
console.log(num3, dec_l);
console.log(num4, dec_r);
```

```
› 5,5
› 5,4
› 3,3
› 3,4
```

The Unary Plus (+) and Negation (-) operators

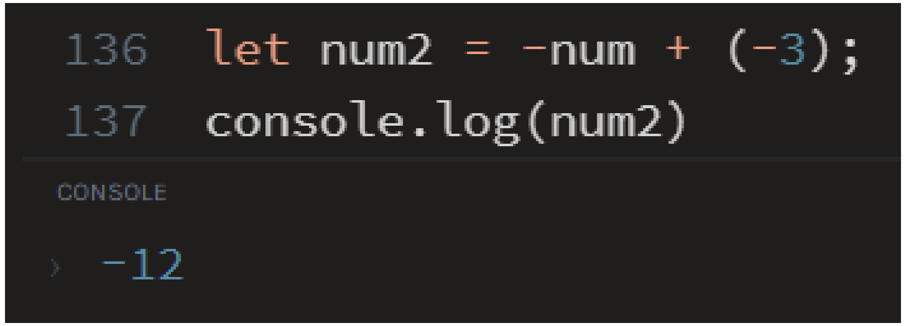
- The unary plus operator is used to try to coerce a value into a number. If it fails, the result is null or NaN.
- E.g., 3 + “6” returns string “36” in JavaScript.
- If you want to “force” the addition, you must coerce “6” into its numeric value.
- The unary negation operator negates the current sign of the number or the variable (positive or negative).
- Note that in these examples, the parenthesis are not necessary but add clarity.



```
let str = 3 + "6";
let num = 3 + (+"6");
console.log(str);
console.log(num);
```

CONSOLE

```
› 36
› 9
```



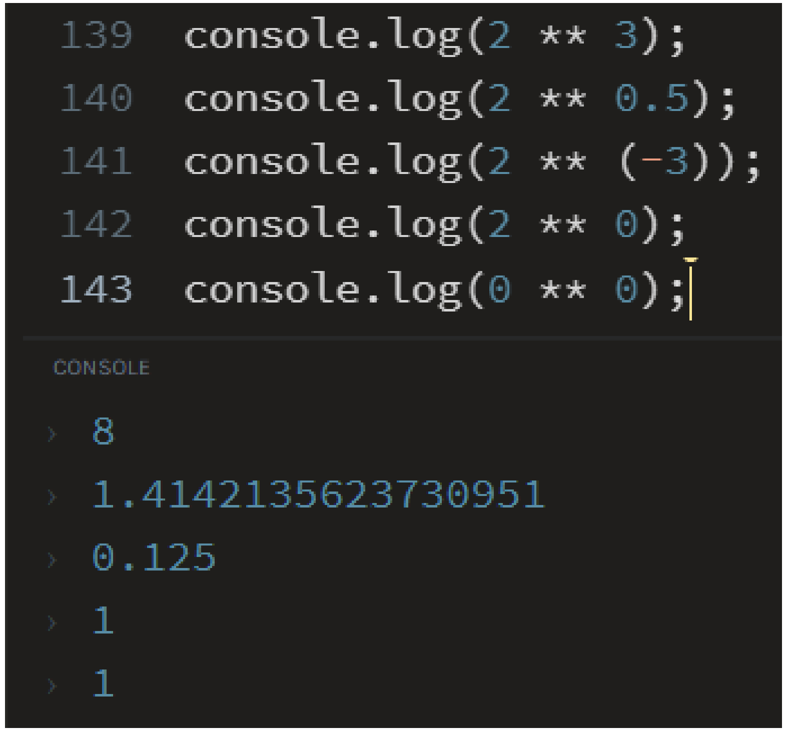
```
let num2 = -num + (-3);
console.log(num2)
```

CONSOLE

```
› -12
```

The Exponentiation operator (**)

- The exponentiation operator raises the first operand to the power of the second operand.
- For recall, the second operand is not necessarily a positive integer (see the second and third examples, square root of 2 and 1/8th).

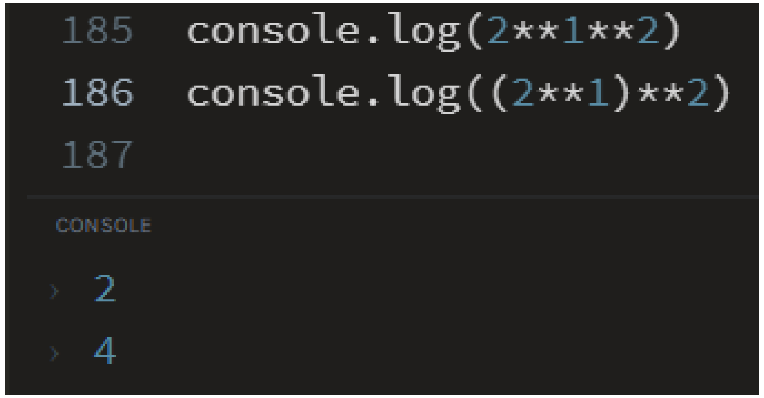


```
console.log(2 ** 3);
console.log(2 ** 0.5);
console.log(2 ** (-3));
console.log(2 ** 0);
console.log(0 ** 0);
```

CONSOLE

```
› 8
› 1.4142135623730951
› 0.125
› 1
› 1
```

- Careful: exponentiations are evaluated from right to left.



```
console.log(2**1**2)
console.log((2**1)**2)
```

CONSOLE

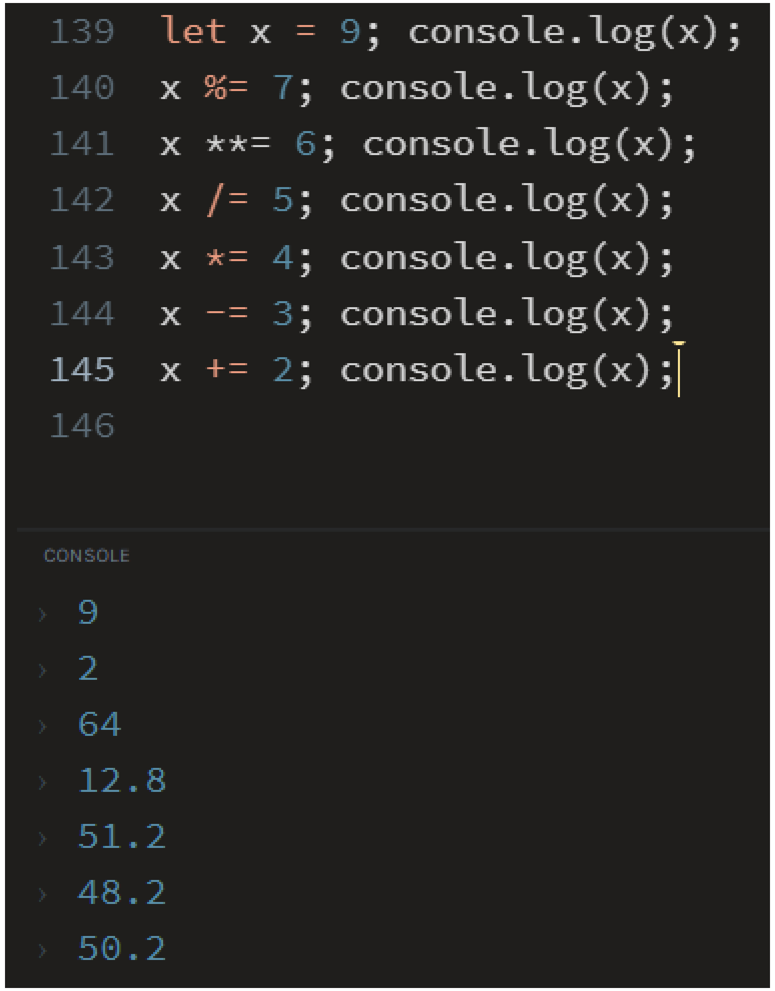
```
› 2
› 4
```

Assignment Operators

Operator	Symbol	Function	Example (with x = 7)
Assignment	=	Assigns the value on the right side of the operator to a variable	x = 7 assigns 7 to x
Add and assign	+=	Adds two values	x += 2 assigns 9 to x
Subtract and assign	-=	Substracts one value from another	x -= 2 assigns 5 to x
Multiply and assign	*=	Multiplies two values	x *= 2 assigns 14 to x
Divide and assign	/=	Divides one value by another	x /= 2 assigns 3.5 to x
Modulus and assign	%=	Returns the remainder	x %= 2 assigns 1 to x
Exponent and assign	**=	Power of a number by another	x **= 2 assigns 49 to x
Bitwise assignments	<<= >>= >>>= &= ^= |=	Performs various bitwise assignments. Out of scope.	

Recall about assignment operators

- Assignment operators assign a value to a variable.
- **They do not compare two items.**
- **They do not perform logical tests.**
- Let **Aop** be an arithmetic operator and x be an **initialised** variable:
    - x **Aop=** 2 is equivalent to x **=** x **Aop** 2.
    - E.g., x += 2 is equivalent to x = x + 2.



```
let x = 9; console.log(x);
x %= 7; console.log(x);
x **= 6; console.log(x);
x /= 5; console.log(x);
x *= 4; console.log(x);
x -= 3; console.log(x);
x += 2; console.log(x);
```

```
9
2
64
12.8
51.2
48.2
50.2
```

Comparison Operators

Operator	Symbol	Function	Example (with x = 7)
Is equal to	==	Returns true if the values on both sides of the operator are equal to each other	x == 7 returns true
Is not equal to	!=	Returns true if the values on both sides of the operator are not equal to each other	x != 6 returns true
Strict is equal to	===	Returns true if the values on both sides of the operator are equal and of the same type	x === 7 returns true
Strict is not equal to	!==	Returns true if the values on both sides of the operator are not equal or not of the same type	x !== “7” returns true
Is greater than	>	Returns true if the value on the left side of the operator is greater than the value on the right side	x > 6 returns true
Is less than	<	Returns true if the value on the left side of the operator is less than the value on the right side	x < 8 returns true
Is greater than or equal to	>=	Returns true if the value on the left side of the operator is greater than or equal to the value on the right side	x >= 7 returns true
Is less than or equal to	<=	Returns true if the value on the left side of the operator is less than or equal to the value on the right side	x <= 7 returns true

Recall about assignment operators

- Assignment operators assign a value to a variable.
- **They do not compare two items.**
- **They do not perform logical tests.**
- Let **Aop** be an arithmetic operator and x be an **initialised** variable:
  - x **Aop=** 2 is equivalent to x **=** x **Aop** 2.
  - E.g., x += 2 is equivalent to x = x + 2.

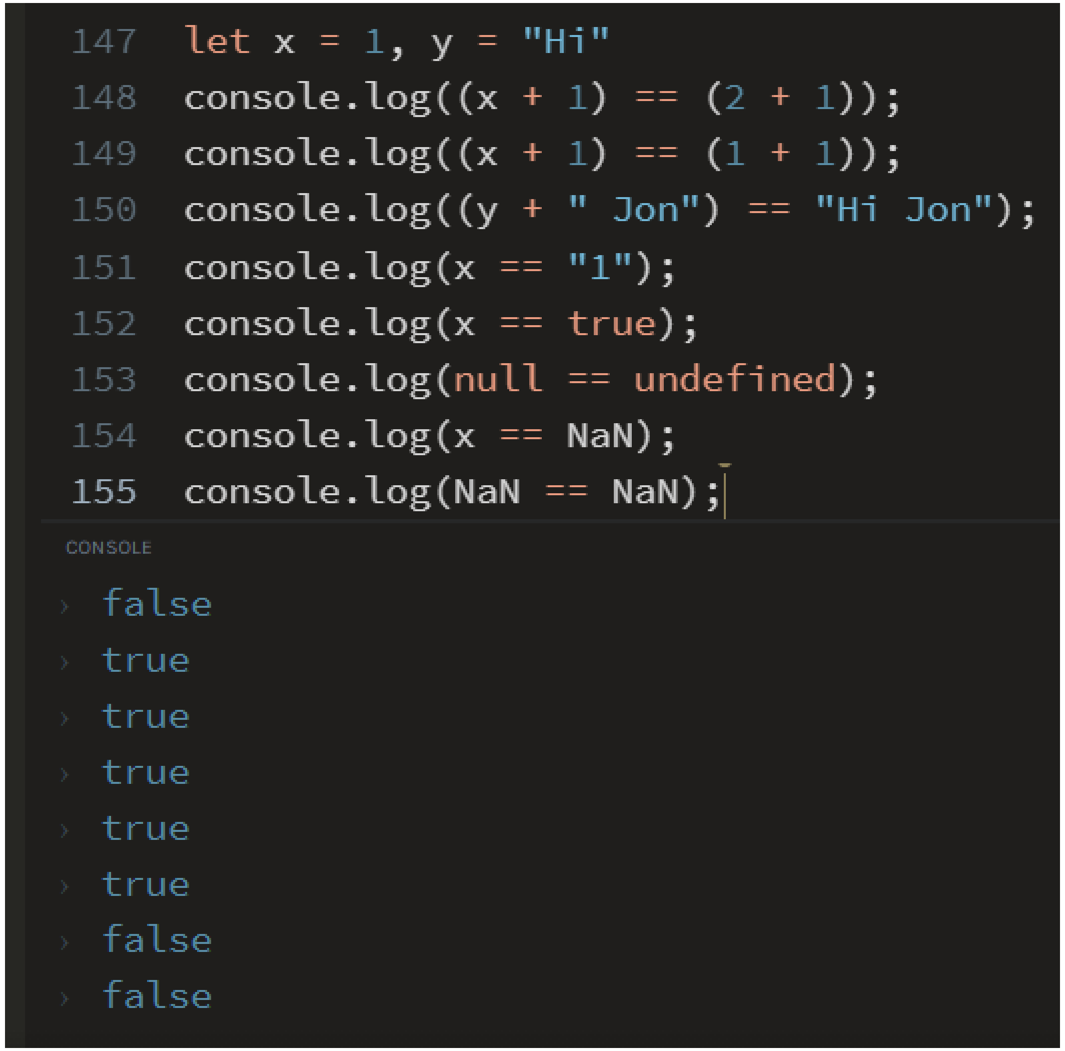
The Is-Equal-To operator (==)

Special rules:

- If one operand is a number and the other a string, it will try to convert the string into a number before testing for equality.
- Same thing with Booleans: true is converted to 1 and false to 0.
- If one operand is null and the other is undefined, it returns true.
- If one or both of the operands is NaN, it returns false.

Advice:

- Use parenthesis to improve the readability.
- Be careful not to mix up the = operator and the == operator.



```
let x = 1, y = "Hi"
console.log((x + 1) == (2 + 1));
console.log((x + 1) == (1 + 1));
console.log((y + " Jon") == "Hi Jon");
console.log(x == "1");
console.log(x == true);
console.log(null == undefined);
console.log(x == NaN);
console.log(NaN == NaN);
```

CONSOLE

```
› false
› true
› true
› true
› true
› true
› false
› false
```

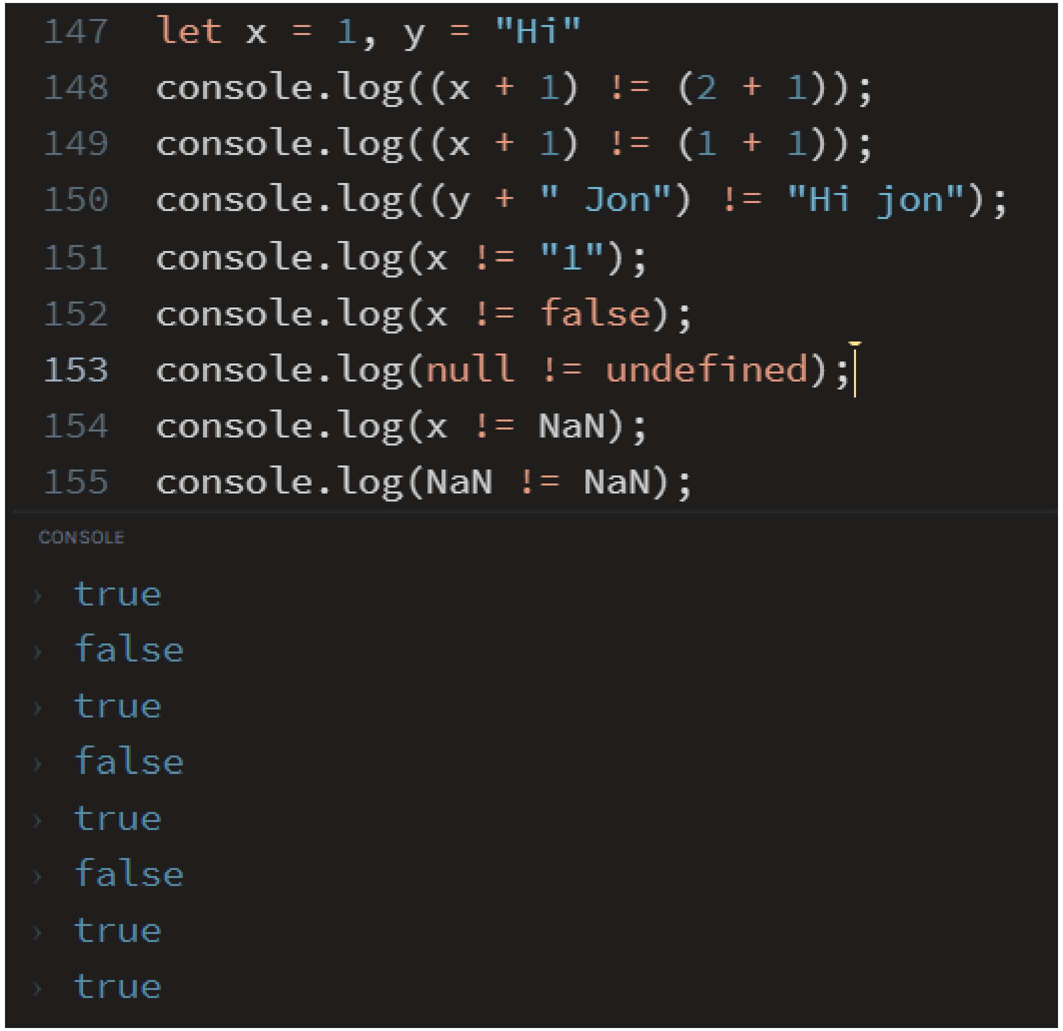
The Is-Not-Equal-To operator (!=)

Special rules:

- If one operand is a number and the other a string, it will try to convert the string into a number before testing for equality.
- Same thing with Booleans: true is converted to 1 and false to 0.
- If one operand is null and the other is undefined, it returns false.
- If one or both of the operands is NaN, it returns true.

Advice:

- Use parenthesis to improve the readability.
- Remember that strings are case sensitive.

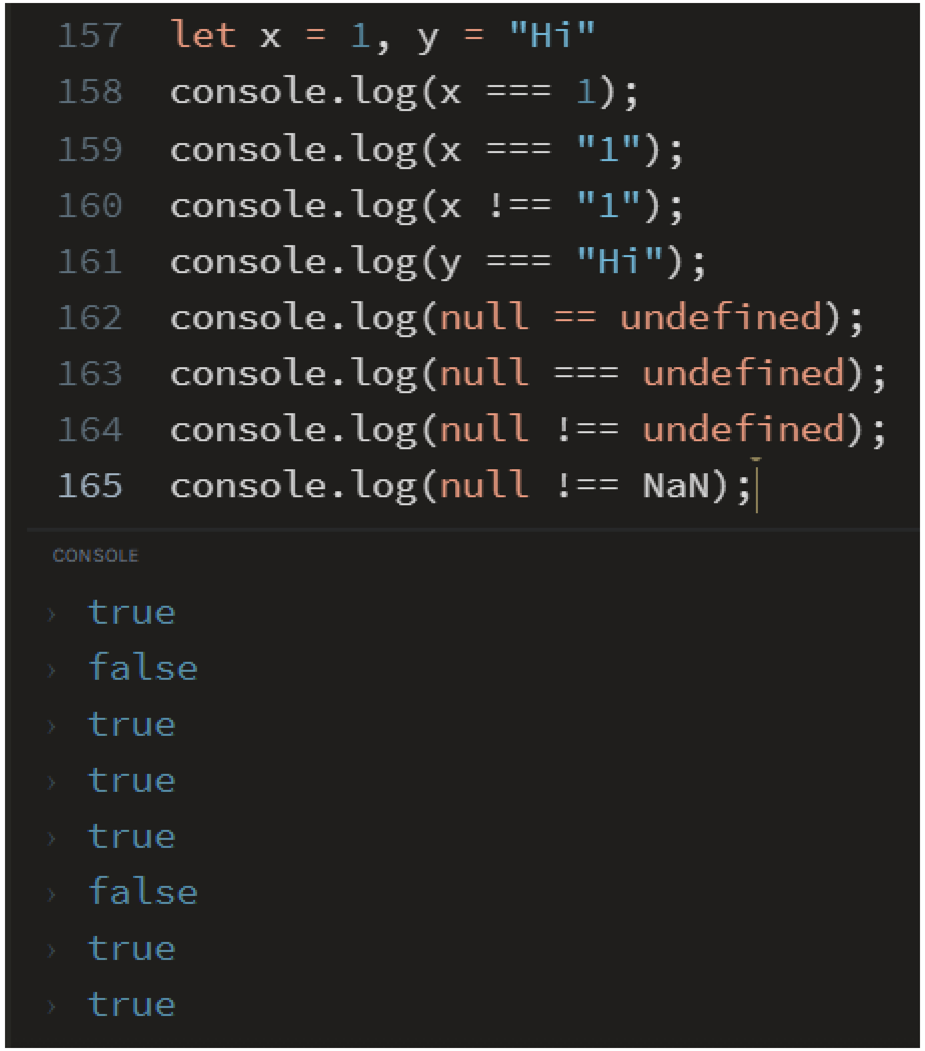


```
let x = 1, y = "Hi"
console.log((x + 1) != (2 + 1));
console.log((x + 1) != (1 + 1));
console.log((y + " Jon") != "Hi jon");
console.log(x != "1");
console.log(x != false);
console.log(null != undefined);
console.log(x != NaN);
console.log(NaN != NaN);
```

```
› true
› false
› true
› false
› true
› false
› true
› true
```

The Strict Is and Is-Not Equal-To operators (=== and !==)

- No type coercion is done with these operators. So,
- For === operator to return true, the operands on each side must be equal and be of the same type.
- For !== operator to return true, the operands on each side must not be equal or must not be of the same type.
- If one operand is null and the other is undefined, === returns false.



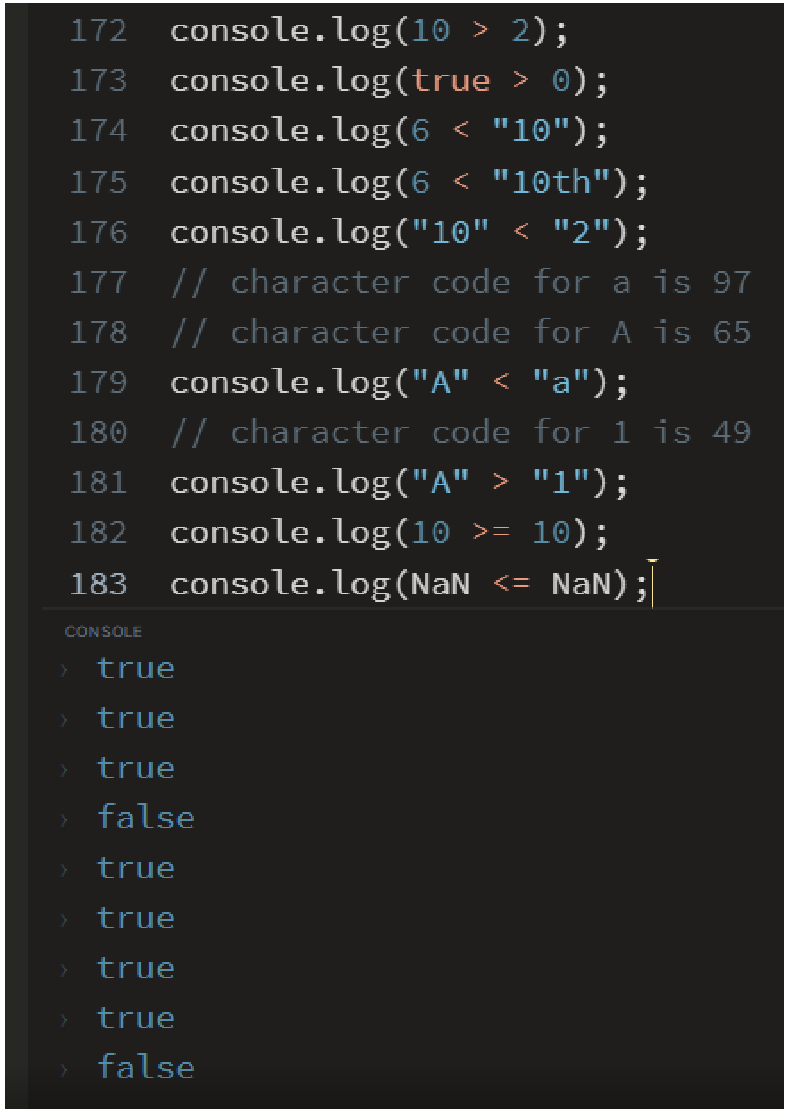
```
let x = 1, y = "Hi"
console.log(x === 1);
console.log(x === "1");
console.log(x !== "1");
console.log(y === "Hi");
console.log(null == undefined);
console.log(null === undefined);
console.log(null !== undefined);
console.log(null !== NaN);
```

```
› true
› false
› true
› true
› true
› false
› true
› true
```

The Is-Greater-Than operator (>) & co. (<, >=, <=)

Special rules:

- If one operand is a number and the other a string, it will first try to convert the string into a number. If it fails, it will convert the number into a string. Then, it will test for comparison.
- Same thing with Booleans: true is converted to 1 and false to 0.
- If both operands are strings, then the strings are compared by their character ASCII codes.
- If one or both of the operands is NaN, it returns false.



```
console.log(10 > 2);
console.log(true > 0);
console.log(6 < "10");
console.log(6 < "10th");
console.log("10" < "2");
// character code for a is 97
// character code for A is 65
console.log("A" < "a");
// character code for 1 is 49
console.log("A" > "1");
console.log(10 >= 10);
console.log(NaN <= NaN);
```

```
true
true
true
false
true
true
true
true
false
```

Logical Operators

Operator	Symbol	Function	Example
AND	&&	Returns true if the statements on both sides of the operator are true	(1 == “1”) && (3 >= 2) returns true (1 === “1”) && (3 >= 2) returns false
OR	||	Returns true if a statement on either side of the operator is true	(3 > 3) || (7 < 9) returns true (3 == 3) || (7 < 9) returns true (3 > 3) || (7 > 9) returns false
NOT	!	Returns true if the statement to the right side of the operator is not true	!(1 === “1”) returns true !(2 < 5) returns false

Bitwise Operators

- Bitwise operators are logical operators that work at the bit level (when every object is represented as a sequence of 0s and 1s.)
- Out of scope but good to know they exist and to be able to spot them.

Operator	Symbol
AND	&
XOR	^
OR	|
NOT	~
Left Shift	<<
Signed Right Shift	>>
Unsigned Right Shift	>>>

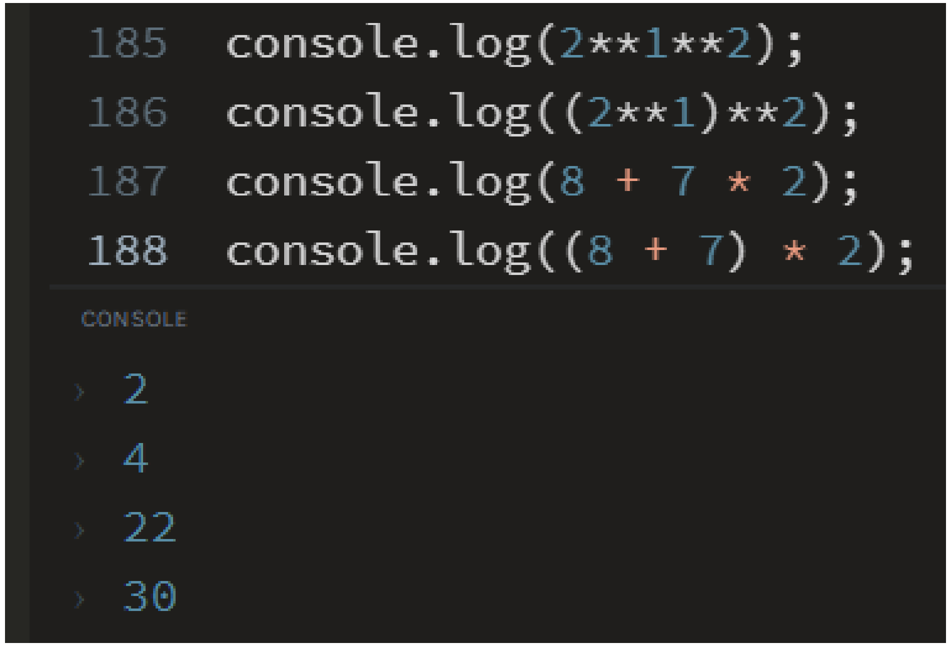
Special Operators

- Here for the sake of completeness.
- Most of these operators are used on concepts that will be covered later. The others will be out of scope.

Operator	Symbol	Function
Conditional	?:	Often used as a short if/else type of statement
Comma	,	Evaluates the statements on both sides of the operator and returns the value of the second statement
Delete	delete	Used to delete an object, a property, or an element in an array
In	in	Returns true if a property is in a specified object
Instanceof	instanceof	Returns true if an object is of a specified object type
New	new	Creates an instance of an object
This	this	Refers to the current object
Typeof	typeof	Returns a string that tells you the type of the value being evaluated
Void	void	Allows an expression to be evaluated without returning a value

Operator precedence

- The operators have a certain order of preference. In a statement with more than one operator, some might be executed before another.
- E.g., in mathematics, exponents are calculated before the multiplication and divisions which are calculated before the additions and subtractions.
- To override the order of preference, use the parentheses to set off the portion that should be executed first.
- Parentheses can also add clarity and make some expressions more readable.



```
console.log(2**1**2);
console.log((2**1)**2);
console.log(8 + 7 * 2);
console.log((8 + 7) * 2);
```

CONSOLE

```
› 2
› 4
› 22
› 30
```

Operator precedence

Type of Operator	Example of operators
1.Parentheses (overrides others)	()
2.Unary	- ++ -- ! ~ typeof void delete
3.Exponent	**
4.Multiplication, division, modulus	* / %
5.Addition, subtraction	+ -
6.Shifts (bitwise)	>>> >> <<
7.Relational comparison	> >= < <= in instanceof
8.Equality comparison	== != === !==
9.AND (bitwise)	&
10.XOR (bitwise)	^
11.OR (bitwise)	|
12.AND (logical)	&&
13.OR (logical)	||
14.Conditional	?:
15.Assignment	= += -= *= /= %= <<= >>= >>>= &= ^= |=
16.Comma	,

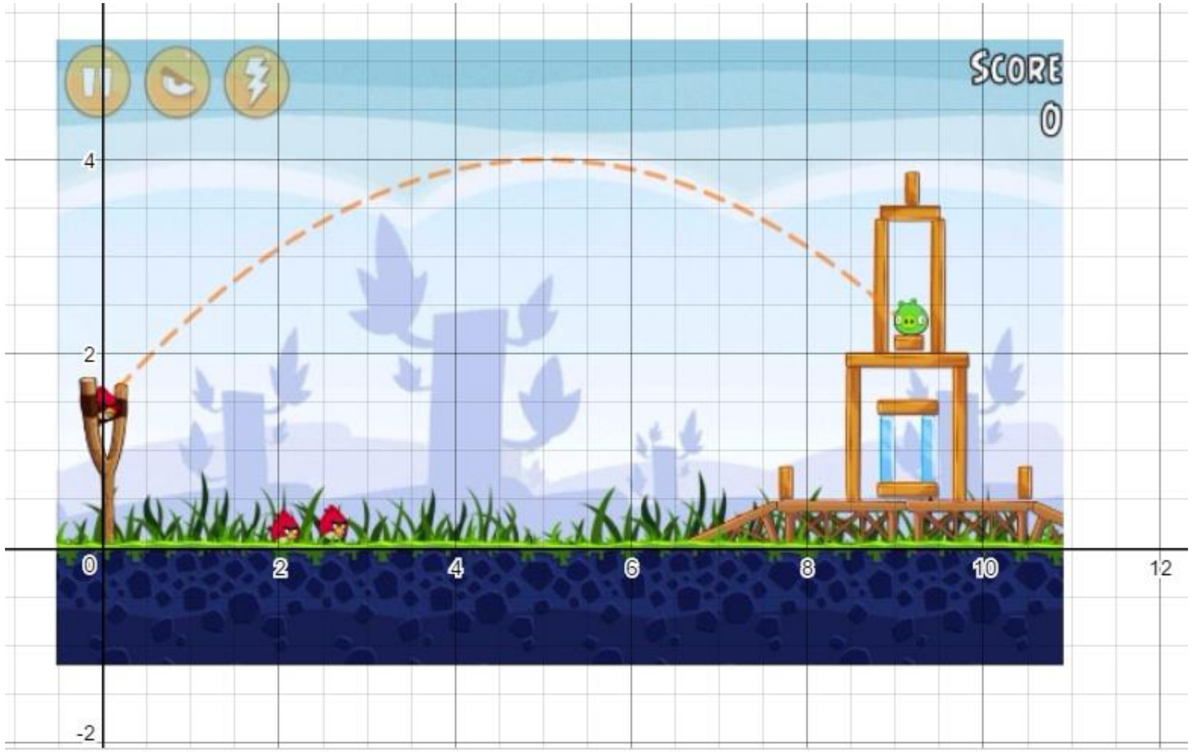
Table. Operator Precedence, from Highest to Lowest

Conclusion

- Functions
- Global and local scopes
- Operators

Activity 2: Ballistics of an angry bird

Let us practice the concepts with a second activity.



Activity 2: Ballistics of an angry bird

Let us practice the concepts with a second activity.

You have to write a single function,

- which computes the distance d at which an angry bird will be landing,
- depending on a given initial angle b,
- and an initial speed a.

Activity 2: Ballistics of an angry bird

Problem statement

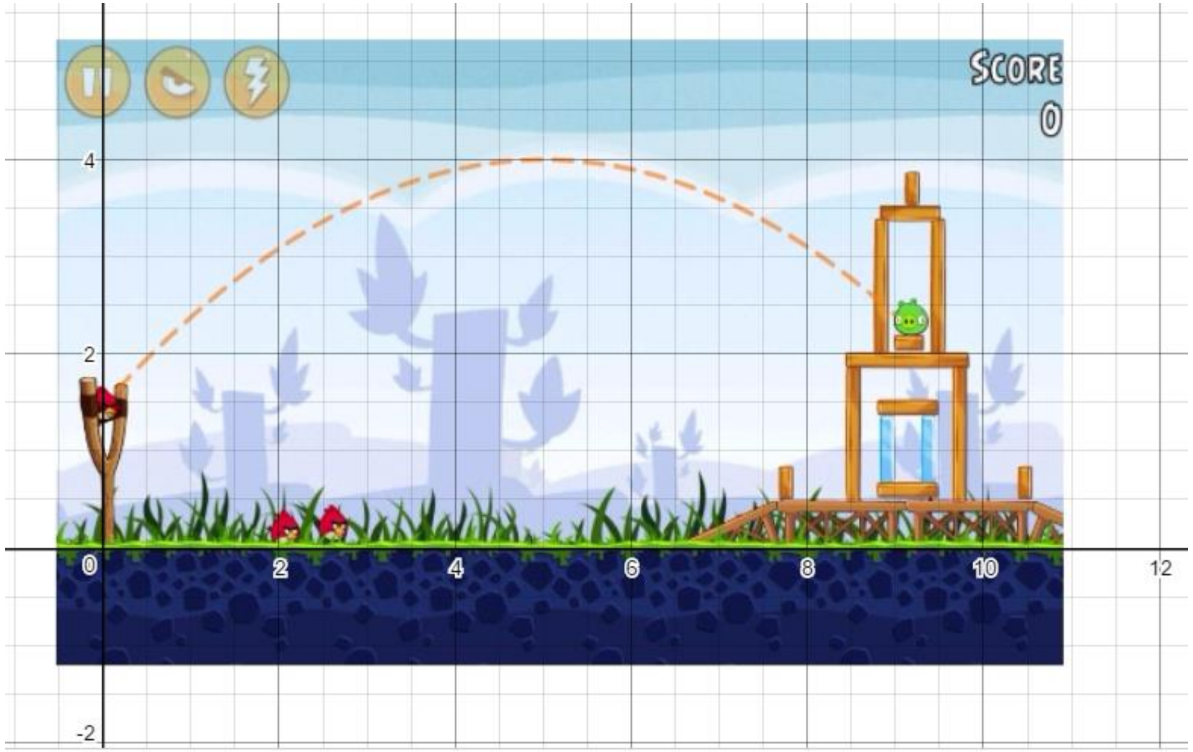
In the angry bird game, the player has to launch birds at structures from a slingshot. The player gets to decide on an initial angle theta (in degrees) and an initial speed for the bird alpha (in m/s).

After releasing, the angry bird goes flying into a parabolic curve, as shown in the figure.

Using Physics, you can compute the distance d, in meters, at which the bird will land as

d = a^2 * sin(2b) / g,

with g the acceleration due to gravity, which we will here set to 9.81 m.s^-2 (same value as the one on Earth!).



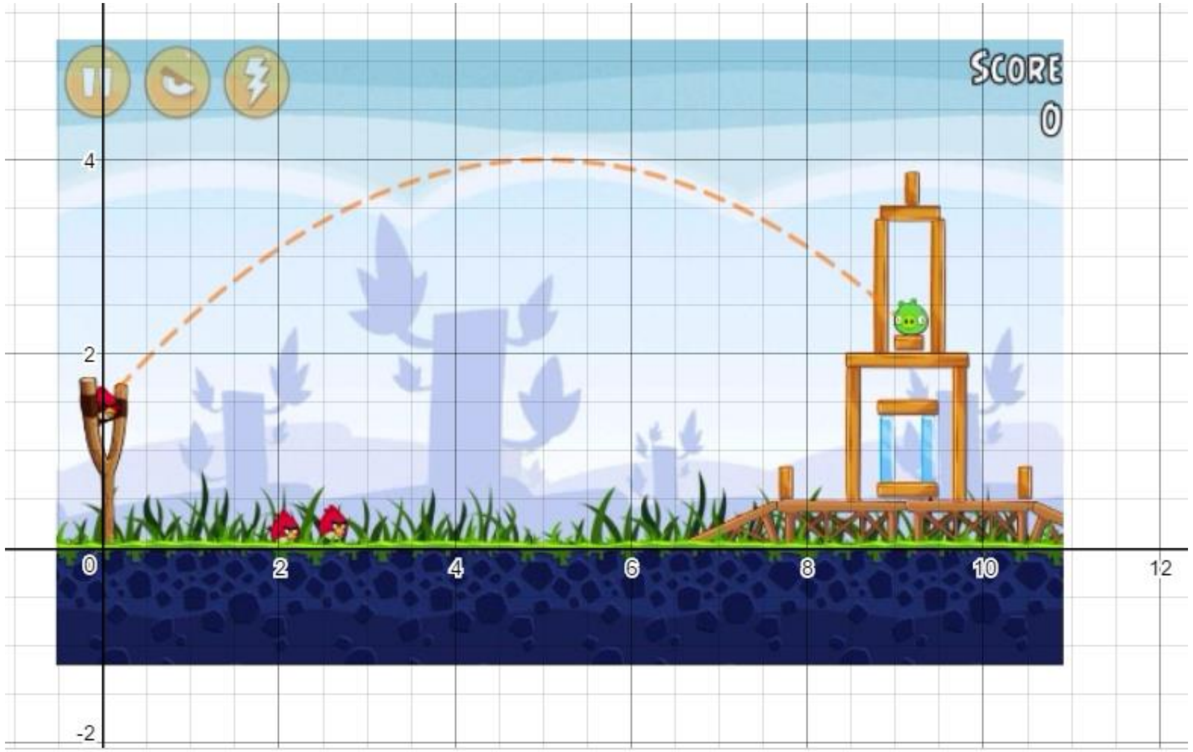
Activity 2: Ballistics of an angry bird

Problem statement

Write a function, compute_landing_point(), which receives the initial angle value b and the initial speed value a, and returns the distance d at which the angry bird will be landing.

Note: You can use the function Math.sin() for sinus.

Important: the angles b will be given in degrees, but most cosine functions in JS require the angle to be in radians. Remember to convert your angles from degrees to radians before using the cosine function! (the conversion ratio is 180° = π)


SCORE
0
4
2
-2
0
2
4
6
8
10
12