**JAVA**

# II. Var

var myName = ‘Arya’;

console.log(myName);

**var** – tạo biến mới (lệnh khác: **let**, **const**)

myName – tên biến *(“camel casing”)*

*After the variable is declared, the string value 'Arya' is printed to the console by referencing the variable name: console.log(myName).*

**There are a few general rules for naming variables:**

* Variable names cannot start with numbers.
* Variable names are case sensitive, so myName and myname would be different variables. It is bad practice to create two variables that have the same name using different cases.
* Variable names cannot be the same as keywords. For a comprehensive list of keywords check out MDN’s keyword documentation.

**Create a Variable: let**

As mentioned in the previous exercise, the let keyword was introduced in ES6. The let keyword signals that the variable can be reassigned a different value. Take a look at the example:

let meal = 'Enchiladas';

console.log(meal); // Output: Enchiladas

meal = 'Burrito';

console.log(meal); // Output: Burrito

**Create a Variable: const**

The **const** keyword was also introduced in ES6, and is short for the word constant. Just like with **var** and **let** you can store any value in a const variable. The way you declare a const variable and assign a value to it follows the same structure as let and var. Take a look at the following example:

const myName = 'Gilberto';

console.log(myName); // Output: Gilberto

However, a const variable cannot be reassigned because it is constant. If you try to reassign a const variable, you’ll get a **TypeError**.

**Mathematical Assignment Operators**

Let’s consider how we can use variables and math operators to calculate new values and assign them to a variable. Check out the example below:

let w = 4;

w = w + 1;

console.log(w); // Output: 5

**or**

let w = 4;

w += 1;

console.log(w); // Output: 5

We also have access to other mathematical assignment operators: -=, \*=, and /= which work in a similar fashion.

**The Increment and Decrement Operator**

Other mathematical assignment operators include the increment operator (++) and decrement operator (--).

The increment operator will increase the value of the variable by 1. The decrement operator will decrease the value of the variable by 1.

**String Concatenation with Variables**

The + operator can be used to combine two string values even if those values are being stored in variables:

let myPet = 'armadillo';

console.log('I own a pet ' + myPet + '.');

// Output: 'I own a pet armadillo.'

**String Interpolation**

In the ES6 version of JavaScript, we can insert, or interpolate, variables into strings using template literals. Check out the following example where a template literal is used to log strings together:

const myPet = 'armadillo';

console.log(`I own a pet ${myPet}.`);

// Output: I own a pet armadillo.

Notice that:

* A template literal is wrapped by backticks ` (this key is usually located on the top of your keyboard, left of the 1 key).
* Inside the template literal, you’ll see a placeholder, ${mypet}. The value of mypet is inserted into the template literal.
* When we interpolate `i own a pet ${mypet}.`, the output we print is the string: 'i own a pet armadillo.'

One of the biggest benefits to using template literals is the readability of the code. Using template literals, you can more easily tell what the new string will be. You also don’t have to worry about escaping double quotes or single quotes.

console.log(`My name is ${myName}. My favorite city is ${myCity}.`);

**typeof operator**

While writing code, it can be useful to keep track of the data types of the variables in your program. If you need to check the data type of a variable’s value, you can use the typeof operator.

The typeofoperator checks the value to its right and returns, or passes back, a string of the data type.

const unknown1 = 'foo';

console.log(typeof unknown1); // Output: string

const unknown2 = 10;

console.log(typeof unknown2); // Output: number

const unknown3 = true;

console.log(typeof unknown3); // Output: boolean

# III. CONDITIONAL STATEMENTS

**What are Conditional Statements?**

In this lesson we will explore how programs make decisions by evaluating conditions and introduce logic into our code! We’ll be covering the following concepts:

* if, else if, and else statements.
* comparison operators.
* logical operators.
* truthy vs falsy values.
* ternary operators.
* the switch statement.

**The if keyword**

if (**điều kiện**) {

console.log('This message will print!');

}

// Prints "This message will print!"

Notice in the example above, we have an if statement. The if statement is composed of:

* The if keyword followed by a set of parentheses () which is followed by a code block, or block statement, indicated by a set of curly braces {}.
* Inside the parentheses (), a condition is provided that evaluates to true or false.
* If the condition evaluates to true, the code inside the curly braces {} runs, or executes.
* If the condition evaluates to false, the block won’t execute.

**If...Else Statements**

if (false) {

console.log('The code in this block will not run.');

} else {

console.log('But the code in this block will!');

}

// Prints "But the code in this block will!"

**Comparison Operators**

When writing conditional statements, sometimes we need to use different types of operators to compare values. These operators are called comparison operators.

Here is a list of some handy comparison operators and their syntax:

* Less than: <
* Greater than: >
* Less than or equal to: <=
* Greater than or equal to: >=
* Is equal to: ===
* Is NOT equal to: !==

We can also use comparison operators on different data types like strings:

'apples' === 'oranges' // false

In the example above, we’re using the identity operator (===) to check if the string 'apples' is the same as the string 'oranges'. Since the two strings are not the same, the comparison statement evaluates to false.

**Logical Operators**

Working with conditionals means that we will be using booleans, true or false values. In JavaScript, there are operators that work with boolean values known as logical operators. We can use logical operators to add more sophisticated logic to our conditionals. There are three logical operators:

* the and operator (&&)
* the or operator (||)
* the not operator, otherwise known as the bang operator (!)

if (stopLight === 'green' && pedestrians === 0) {

console.log('Go!');

} else {

console.log('Stop');

}

**Truthy and Falsy**

Let’s consider how non-boolean data types, like strings or numbers, are evaluated when checked inside a condition.

Sometimes, you’ll want to check if a variable exists and you won’t necessarily want it to equal a specific value— you’ll only check to see if the variable has been assigned a value.

Here’s an example:

let myVariable = 'I Exist!';

if (myVariable) {

console.log(myVariable)

} else {

console.log('The variable does not exist.')

}

The code block in the if statement will run because myVariable has a truthy value; even though the value of myVariable is not explicitly the value true, when used in a boolean or conditional context, it evaluates to true because it has been assigned a non-falsy value.

So which values are falsy— or evaluate to false when checked as a condition? The list of falsy values includes:

* 0
* Empty strings like "" or ''
* null which represent when there is no value at all
* **undefined** which represent when a declared variable lacks a value
* **NaN**, or Not a Number

**Truthy and Falsy Assignment**

Say you have a website and want to take a user’s username to make a personalized greeting. Sometimes, the user does not have an account, making the username variable falsy. The code below checks if username is defined and assigns a default string if it is not:

let defaultName;

if (username) {

defaultName = username;

} else {

defaultName = 'Stranger';

}

If you combine your knowledge of logical operators you can use a short-hand for the code above. In a boolean condition, JavaScript assigns the truthy value to a variable if you use the || operator in your assignment:

let defaultName = username || 'Stranger';

Because || or statements check the left-hand condition first, the variable defaultName will be assigned the actual value of username if is truthy, and it will be assigned the value of 'Stranger' if username is falsy. This concept is also referred to as short-circuit evaluation.

**Ternary Operator**

In the spirit of using short-hand syntax, we can use a ternary operator to simplify an if...else statement.

Take a look at the if...else statement example:

let isNightTime = true;

if (isNightTime) {

console.log('Turn on the lights!');

} else {

console.log('Turn off the lights!');

}

We can use a ternary operator to perform the same functionality:

isNightTime ? console.log('Turn on the lights!') : console.log('Turn off the lights!');

In the example above:

* The condition, isNightTime, is provided before the ?.
* Two expressions follow the ? and are separated by a colon :.
* If the condition evaluates to true, the first expression executes.
* If the condition evaluates to false, the second expression executes.

Like if...else statements, ternary operators can be used for conditions which evaluate to true or false.

**Else If Statements**

We can add more conditions to our if...else with an else if statement.

let stopLight = 'yellow';

if (stopLight === 'red') {

console.log('Stop!');

} else if (stopLight === 'yellow') {

console.log('Slow down.');

} else if (stopLight === 'green') {

console.log('Go!');

} else {

console.log('Caution, unknown!');

}

**The switch keyword**

else if statements are a great tool if we need to check multiple conditions. In programming, we often find ourselves needing to check multiple values and handling each of them differently. For example:

let groceryItem = 'papaya';

if (groceryItem === 'tomato') {

console.log('Tomatoes are $0.49');

} else if (groceryItem === 'papaya'){

console.log('Papayas are $1.29');

} else {

console.log('Invalid item');

}

In the code above, we have a series of conditions checking for a value that matches a groceryItem variable. Our code works fine, but imagine if we needed to check 100 different values! Having to write that many else if statements sounds like a pain!

A switch statement provides an alternative syntax that is easier to read and write. A switch statement looks like this:

let groceryItem = 'papaya';

switch (groceryItem) {

case 'tomato':

console.log('Tomatoes are $0.49');

break;

case 'lime':

console.log('Limes are $1.49');

break;

case 'papaya':

console.log('Papayas are $1.29');

break;

default:

console.log('Invalid item');

break;

}

// Prints 'Papayas are $1.29'

* The switch keyword initiates the statement and is followed by ( ... ), which contains the value that each case will compare. In the example, the value or expression of the switch statement is groceryItem.
* Inside the block, { ... }, there are multiple cases. The case keyword checks if the expression matches the specified value that comes after it. The value following the first case is 'tomato'. If the value of groceryItem equalled 'tomato', that case‘s console.log() would run.
* The value of groceryItem is 'papaya', so the third case runs— Papayas are $1.29 is logged to the console.
* The break keyword tells the computer to exit the block and not execute any more code or check any other cases inside the code block. Note: Without the break keyword at the end of each case, the program would execute the code for all matching cases and the default code as well. This behavior is different from if/else conditional statements which execute only one block of code.
* At the end of each switch statement, there is a default statement. If none of the cases are true, then the code in the default statement will run.

# III. FUNCTIONS

**What are functions?**

In programming, we often use code to perform a specific task multiple times. Instead of rewriting the same code, we can group a block of code together and associate it with one task, then we can reuse that block of code whenever we need to perform the task again. We achieve this by creating a function. A function is a reusable block of code that groups together a sequence of statements to perform a specific task.

In this lesson, you will learn how to create and use functions, and how they can be used to create clearer and more concise code.

**Function Declarations**

Take a look at example of hoisting:

console.log(greetWorld()); // Output: Hello, World!

function greetWorld() {

console.log('Hello, World!');

}

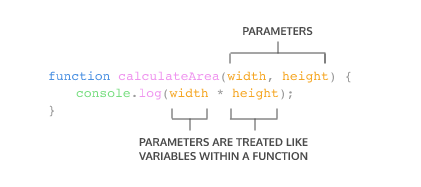
Notice how hoisting allowed greetWorld() to be called before the greetWorld() function was defined! Since hoisting isn’t considered good practice, we simply want you to be aware of this feature.

**Calling a Function**

To call a function in your code, you type the function name followed by parentheses.

This function call executes the function body, or all of the statements between the curly braces in the function declaration.

**Parameters and Arguments**



In the diagram above, calculateArea(), computes the area of a rectangle, based on two inputs, width and height. The parameters are specified between the parenthesis as width and height, and inside the function body, they act just like regular variables. width and height act as placeholders for values that will be multiplied together.

When calling a function that has parameters, we specify the values in the parentheses that follow the function name. The values that are passed to the function when it is called are called arguments. Arguments can be passed to the function as values or variables.

**Default Parameters**

Default parameters allow parameters to have a predetermined value in case there is no argument passed into the function or if the argument is undefined when called.

Take a look at the code snippet below that uses a default parameter:

function greeting (name = 'stranger') {

console.log(`Hello, ${name}!`)

}

greeting('Nick') // Output: Hello, Nick!

greeting() // Output: Hello, stranger!

* In the example above, we used the = operator to assign the parameter name a default value of 'stranger'. This is useful to have in case we ever want to include a non-personalized default greeting!
* When the code calls greeting('Nick') the value of the argument is passed in and, 'Nick', will override the default parameter of 'stranger' to log 'Hello, Nick!' to the console.
* When there isn’t an argument passed into greeting(), the default value of 'stranger' is used, and 'Hello, stranger!' is logged to the console.

By using a default parameter, we account for situations when an argument isn’t passed into a function that is expecting an argument.

**Return**

When a function is called, the computer will run through the function’s code and evaluate the result of calling the function. By default that resulting value is undefined.

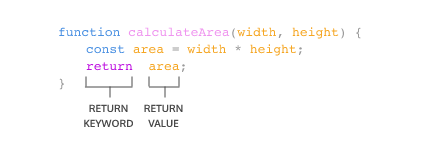
function rectangleArea(width, height) {

let area = width \* height

}

console.log(rectangleArea(5, 7)) // Prints undefined

In the code example, we defined our function to calculate the area of a width and height parameter. Then rectangleArea() is invoked with the arguments 5 and 7. But when we went to print the results we got undefined. Did we write our function wrong? No! In fact, the function worked fine, and the computer did calculate the area as 35, but we didn’t capture it. So how can we do that? With the keyword return!



To pass back information from the function call, we use a return statement. To create a return statement, we use the return keyword followed by the value that we wish to return. Like we saw above, if the value is omitted, undefined is returned instead.

When a return statement is used in a function body, the execution of the function is stopped and the code that follows it will not be executed. Look at the example below:

function rectangleArea(width, height) {

if (width < 0 || height < 0) {

return 'You need positive integers to calculate area!';

}

return width \* height;

}

If an argument for width or height is less than 0, then rectangleArea() will return 'You need positive integers to calculate area!'. The second return statement width \* height will not run.

The return keyword is powerful because it allows functions to produce an output. We can then save the output to a variable for later use.

function monitorCount(rows, columns) {

return rows\*columns;

}

const numOfMonitors = monitorCount(5, 4);

console.log(numOfMonitors);

**Helper Functions**

We can also use the return value of a function inside another function. These functions being called within another function are often referred to as helper functions. Since each function is carrying out a specific task, it makes our code easier to read and debug if necessary.

If we wanted to define a function that converts the temperature from Celsius to Fahrenheit, we could write two functions like:

function multiplyByNineFifths(number) {

return number \* (9/5);

};

function getFahrenheit(celsius) {

return multiplyByNineFifths(celsius) + 32;

};

getFahrenheit(15); // Returns 59

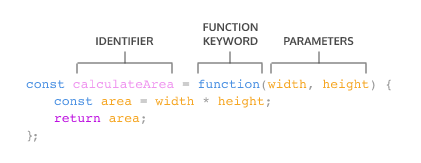
In the example above:

* getFahrenheit() is called and 15 is passed as an argument.
* The code block inside of getFahrenheit() calls multiplyByNineFifths() and passes 15 as an argument.
* multiplyByNineFifths() takes the argument of 15 for the number parameter.
* The code block inside of multiplyByNineFifths() function multiplies 15 by (9/5), which evaluates to 27.
* 27 is returned back to the function call in getFahrenheit().
* getFahrenheit() continues to execute. It adds 32 to 27, which evaluates to 59.
* Finally, 59 is returned back to the function call getFahrenheit(15).

**Function Expressions**

Another way to define a function is to use a function expression. To define a function inside an expression, we can use the function keyword. In a function expression, the function name is usually omitted. A function with no name is called an anonymous function. A function expression is often stored in a variable in order to refer to it.

Consider the following function expression:



To declare a function expression:

1. Declare a variable to make the variable’s name be the name, or identifier, of your function. Since the release of ES6, it is common practice to use const as the keyword to declare the variable.
2. Assign as that variable’s value an anonymous function created by using the function keyword followed by a set of parentheses with possible parameters. Then a set of curly braces that contain the function body.

To invoke a function expression, write the name of the variable in which the function is stored followed by parentheses enclosing any arguments being passed into the function.

variableName(argument1, argument2)

Unlike function declarations, function expressions are not hoisted so they cannot be called before they are defined.

const plantNeedsWater = function(day) {

if (day === 'Wednesday') {

return true

} else {

return false

}

}

console.log(plantNeedsWater('Tuesday'))

**Arrow Functions**

ES6 introduced arrow function syntax, a shorter way to write functions by using the special “fat arrow” () => notation.

Arrow functions remove the need to type out the keyword function every time you need to create a function. Instead, you first include the parameters inside the ( ) and then add an arrow => that points to the function body surrounded in { } like this:

const rectangleArea = (width, height) => {

let area = width \* height;

return area;

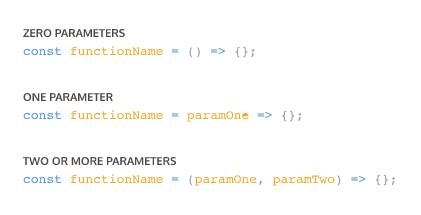
};

It’s important to be familiar with the multiple ways of writing functions because you will come across each of these when reading other JavaScript code.

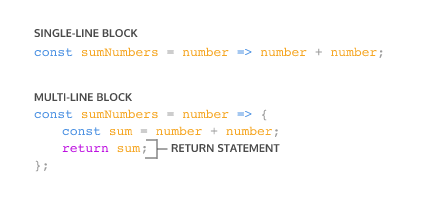
**Concise Body Arrow Functions**

JavaScript also provides several ways to refactor arrow function syntax. The most condensed form of the function is known as concise body. We’ll explore a few of these techniques below:

1. Functions that take only a single parameter do not need that parameter to be enclosed in parentheses. However, if a function takes zero or multiple parameters, parentheses are required.



1. A function body composed of a single-line block does not need curly braces. Without the curly braces, whatever that line evaluates will be automatically returned. The contents of the block should immediately follow the arrow => and the return keyword can be removed. This is referred to as implicit return.



So if we have a function:

const squareNum = (num) => {

return num \* num;

};

We can refactor the function to:

const squareNum = num => num \* num;

Notice the following changes:

* The parentheses around num have been removed, since it has a single parameter.
* The curly braces { } have been removed since the function consists of a single-line block.
* The return keyword has been removed since the function consists of a single-line block.

|  |  |
| --- | --- |
| const plantNeedsWater = (day) => {    if (day === 'Wednesday') {      return true;    } else {      return false;    }  }; | const plantNeedsWater = day => day === 'Wednesday' ? true : false; |

# III. SCOPE

An important idea in programming is scope. Scope defines where variables can be accessed or referenced. While some variables can be accessed from anywhere within a program, other variables may only be available in a specific context.

You can think of scope like the view of the night sky from your window. Everyone who lives on the planet Earth is in the global scope of the stars. The stars are accessible globally. Meanwhile, if you live in a city, you may see the city skyline or the river. The skyline and river are only accessible locally in your city, but you can still see the stars that are available globally.

**Blocks and Scope**

We’ve seen blocks used before in functions and if statements. A block is the code found inside a set of curly braces {}. Blocks help us group one or more statements together and serve as an important structural marker for our code.

A block of code could be a function, like this:

Notice that the function body is actually a block of code.

const logSkyColor = () => {

let color = 'blue';

console.log(color); // blue

};

Observe the block in an if statement:

if (dusk) {

let color = 'pink';

console.log(color); // pink

};

In the next few exercises, we’ll see how blocks define the scope of variables.

⭐

Beneath the logCitySkyline() function, use console.log() to log the value of logCitySkyline() to the console.

You’ll notice that the logCitySkyline() function is able to access both variables without any problems. In the next exercise we’ll consider why would it be preferable to have one variable outside of a block and the other inside of a block.

#solve:

const city = 'New York City';

function logCitySkyline() {

let skyscraper = 'Empire State Building';

return 'The stars over the ' + skyscraper + ' in ' + city;

console.log(logCitySkyline());

}

**Global Scope**

Scope is the context in which our variables are declared. We think about scope in relation to blocks because variables can exist either outside of or within these blocks.

In global scope, variables are declared outside of blocks. These variables are called global variables. Because global variables are not bound inside a block, they can be accessed by any code in the program, including code in blocks.

Let’s take a look at an example of global scope:

const color = 'blue'

const returnSkyColor = () => {

return color; // blue

};

console.log(returnSkyColor()); // blue

* Even though the color variable is defined outside of the block, it can be accessed in the function block, giving it global scope.
* In turn, color can be accessed within the returnSkyColor function block.

⭐

Beneath the callMyNightSky() function, use console.log() to log the value of callMyNightSky() to the console.

You’ll notice that the function block for callMyNightSky() is able to access the global variables freely since the variables are available to all lines of code in the file.

#solve:

const satellite = 'The Moon';

const galaxy = 'The Milky Way';

const stars = 'North Star';

function callMyNightSky() {

return 'Night Sky: ' + satellite + ', ' + stars + ', and ' + galaxy;

console.log(callMyNightSky());

}

**Block Scope**

The next context we’ll cover is block scope. When a variable is defined inside a block, it is only accessible to the code within the curly braces {}. We say that variable has block scope because it is only accessible to the lines of code within that block.

Variables that are declared with block scope are known as local variables because they are only available to the code that is part of the same block.

Block scope works like this:

const logSkyColor = () => {

let color = 'blue';

console.log(color); // blue

};

logSkyColor(); // blue

console.log(color); // ReferenceError

You’ll notice:

* We define a function logSkyColor().
* Within the function, the color variable is only available within the curly braces of the function.
* If we try to log the same variable outside the function, throws a ReferenceError.

**Scope Pollution**

It may seem like a great idea to always make your variables accessible, but having too many global variables can cause problems in a program.

When you declare global variables, they go to the global namespace. The global namespace allows the variables to be accessible from anywhere in the program. These variables remain there until the program finishes which means our global namespace can fill up really quickly.

Scope pollution is when we have too many global variables that exist in the global namespace, or when we reuse variables across different scopes. Scope pollution makes it difficult to keep track of our different variables and sets us up for potential accidents. For example, globally scoped variables can collide with other variables that are more locally scoped, causing unexpected behavior in our code.

Let’s look at an example of scope pollution in practice so we know how to avoid it:

let num = 50;

const logNum = () => {

num = 100; // Take note of this line of code

console.log(num);

};

logNum(); // Prints 100

console.log(num); // Prints 100

You’ll notice:

* We have a variable num.
* Inside the function body of logNum(), we want to declare a new variable but forgot to use the let keyword.
* When we call logNum(), num gets reassigned to 100.
* The reassignment inside logNum() affects the global variable num.
* Even though the reassignment is allowed and we won’t get an error, if we decided to use num later, we’ll unknowingly use the new value of num.

⭐

Outside the function, under the current console.log() statement, add another console.log() statement to log stars to the console.

You’ll notice that the global variable stars was reassigned to 'Sirius'. In other words, we changed the value of the global stars variable but it’s not easy to read what exactly happened. This is bad practice in code maintainability and could impact our program in ways we do not intend.

const satellite = 'The Moon';

const galaxy = 'The Milky Way';

let stars = 'North Star';

const callMyNightSky = () => {

stars = 'Sirius';

return 'Night Sky: ' + satellite + ', ' + stars + ', ' + galaxy;

};

console.log(callMyNightSky());

console.log(stars);

**Practice Good Scoping**

Given the challenges with global variables and scope pollution, we should follow best practices for scoping our variables as tightly as possible using block scope.

Tightly scoping your variables will greatly improve your code in several ways:

* It will make your code more legible since the blocks will organize your code into discrete sections.
* It makes your code more understandable since it clarifies which variables are associated with different parts of the program rather than having to keep track of them line after line!
* It’s easier to maintain your code, since your code will be modular.
* It will save memory in your code because it will cease to exist after the block finishes running.

Here’s another example of how to use block scope, as defined within an if block:

const logSkyColor = () => {

const dusk = true;

let color = 'blue';

if (dusk) {

let color = 'pink';

console.log(color); // pink

}

console.log(color); // blue

};

console.log(color); // ReferenceError

Here, you’ll notice:

* We create a variable dusk inside the logSkyColor() function.
* After the if statement, we define a new code block with the {} braces. Here we assign a new value to the variable color if the if statement is truthy.
* Within the if block, the color variable holds the value 'pink', though outside the if block, in the function body, the color variable holds the value 'blue'.
* While we use block scope, we still pollute our namespace by reusing the same variable name twice. A better practice would be to rename the variable inside the block.

Block scope is a powerful tool in JavaScript, since it allows us to define variables with precision, and not pollute the global namespace. If a variable does not need to exist outside a block— it shouldn’t!

# IV. Arrays

let <arrays name>