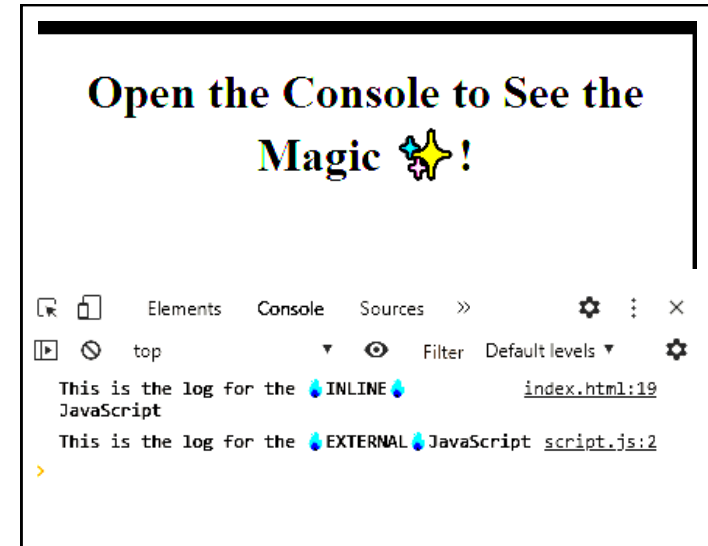


Console Log

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Script Tags and Console Log</title>
</head>
<body>
  <h1 style="text-align:center;">Open the Console to See the Magic ✨! </h1>
  <!--Inline script-->
  <script>
    //Console logs write data directly to the console.
    console.log("This is the log for the ✨INLINE✨ JavaScript");
  </script>
  <!--Link to external JavaScript file-->
  <script src="script.js"></script>
</body>
</html>
```



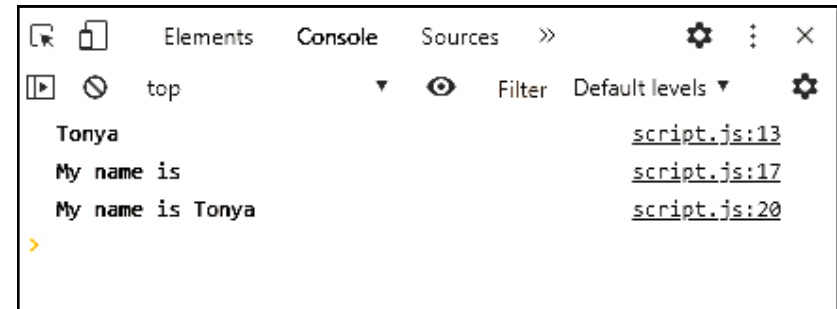
// External files make your code organized and easier to maintain

```
console.log("This is the log for the ✨EXTERNAL✨ JavaScript");
```

Variables

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Hello Variable</title>
</head>
<body>

  <h1 style="text-align:center;">Open the Console to See the Magic ☐! </h1>
  <script src="script.js"></script>
</body>
</html>
```

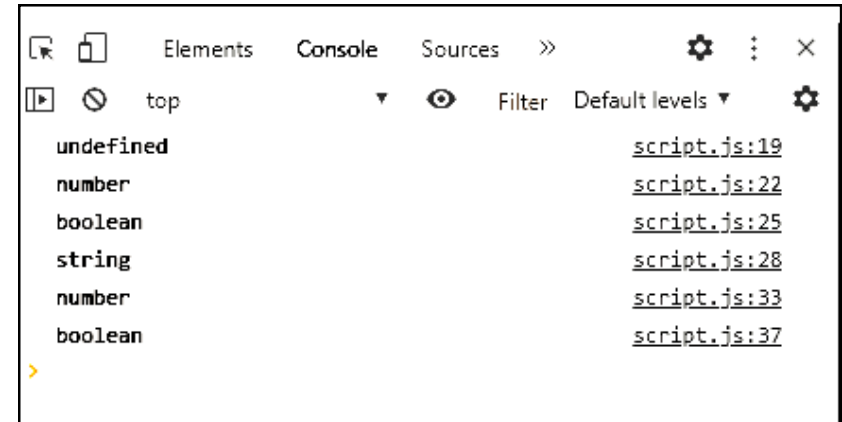


```
// Declares student variable using var keyword
var studentName;
// Uses assignment operator(=) to assign a value
var studentName = "Abdul";
var studentAge = 32;
// To re-assign a variable, use only the variable's name
studentName = "Tonya";
studentAge = 52;
// To access a value stored in a variable, use the variable's name
console.log(studentName);
//To combine the message with a variable value use the concatenation operator(+)
//Logs "My name is "
console.log("My name is ");
// Logs "My name is Tonya"
console.log("My name is " + studentName);
```

Primitives

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Primitive Types</title>
</head>
<body>

  <h1 style="text-align:center;">Open the Console to See the Magic ☐! </h1>
  <script src="script.js"></script>
</body>
</html>
```



//Primitive data types include undefined, string, number and boolean

//Undefined variables haven't been assigned values yet.

```
var myUndefined;
```

// A string is a series of characters and is surrounded by quotes

```
var myStringWelcome = "Hello World";
```

```
var myStringPassword = "865Password!2020";
```

// A number can be either an integer or a decimal

```
var myNumberInt = 100;
```

```
var myNumberDecimal = 100.100;
```

// Booleans have two values: true or false

```
var isMyBooleanTrue = true;
```

```
var isMyBooleanFalse = false;
```

// To check the type of data, use typeof

followed by the name of the variable

// Logs undefined

```
console.log(typeof myUndefined);
```

// Logs number

```
console.log(typeof myNumberInt);
```

// Logs boolean

```
console.log(typeof true);
```

// Logs string

```
console.log(typeof "Howdy");
```

// Pro-tip: JavaScript is loosely typed,

so the type is tied to the value held in the variable,
not the variable itself!

// Logs number

```
var myVariable = 33;
```

```
console.log(typeof myVariable);
```

// myVariable is reassigned; Logs boolean

```
myVariable = false;
```

```
console.log(typeof myVariable);
```

Operators

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Primitive Types</title>
</head>
<body>

  <h1 style="text-align:center;">Open the Console to See the Magic ☐! </h1>
  <script src="script.js"></script>
</body>
</html>
```

```
var a = 100;
var b = 10;
var c = "10";
```

// Arithmetic operators combine with numbers to form an expression
that returns a single number

```
console.log(a + b);
console.log(a - b);
console.log(a / b);
console.log(a * b);
```

// Modulus returns the remainder between two numbers.

```
console.log(a % b);
```

// Comparison operators combine with strings, booleans and numbers
to form an expression that evaluates to true or false

// Compares equality

```
console.log(b == c);
console.log(b != c);
```

// Compares equality and type (strict equality)

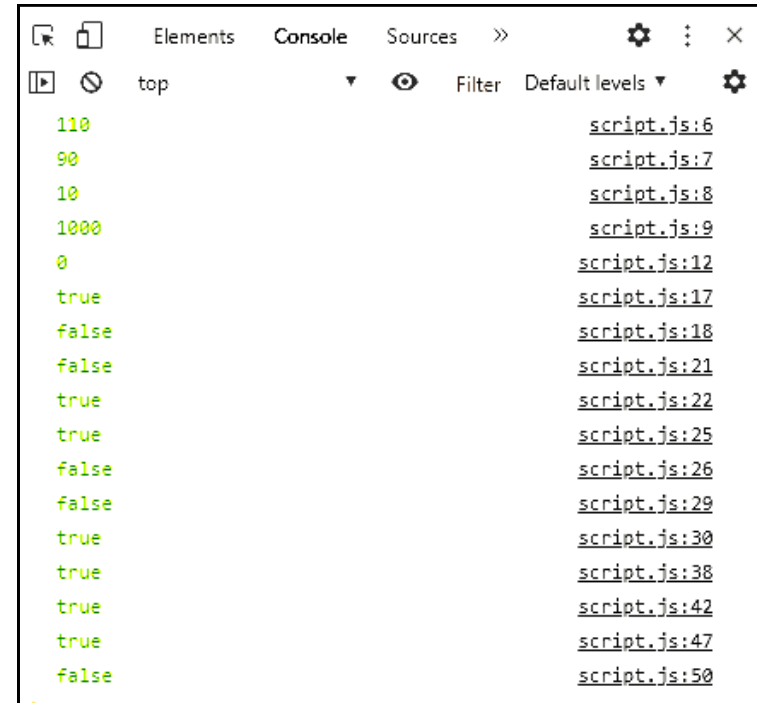
```
console.log(b === c);
console.log(b !== c);
```

// Greater than or less than

```
console.log(a > b);
console.log(a < b);
```

// Greater than or equal to and less than or equal to

```
console.log(a <= b);
console.log(a >= b);
```



// Logical operators take in two or more expressions and return true or false

```
var expression1 = (b == c);
```

```
var expression2 = (a > b);
```

// Evaluates to true if expression1 AND expression2 are both true,
otherwise false

```
console.log(expression1 && expression2);
```

// Evaluates to true if expression1 OR expression2 is true, otherwise false

```
console.log(expression1 || expression2);
```

// Logical Not (!) turns an expression that evaluates to true to false
and vice versa

// Returns true

```
console.log(expression2);
```

// Returns false

```
console.log(!expression2);
```

Conditional Statements

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Conditional Statements</title>
</head>
<body>

  <h1 style="text-align:center;">Open the Console to See the Magic ☐! </h1>
  <script src="script.js"></script>
</body>
</html>
```

```
var hungerLevel = 50;
var isLunchTime = true;
var lunchBill = 11;
```

// If statement

// Evaluates to true so "Hungry" is logged

```
if (hungerLevel >= 50) {
  console.log("Hungry!");
}
```

// Evaluates to false so nothing is logged

```
if (hungerLevel < 50) {
  console.log("Hungry!");
}
```

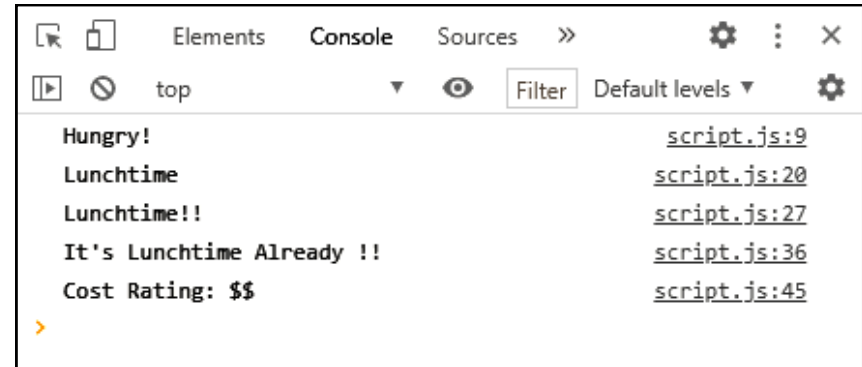
// Else statement

// Evaluates to true so "Lunchtime" is logged

```
if (isLunchTime === true) {
  console.log("Lunchtime");
} else {
  console.log("Not Lunchtime");
}
```

// isLunchTime is another way of writing "isLunchTime === true"

```
if (isLunchTime) {
  console.log("Lunchtime!!");
} else {
  console.log("Not Lunchtime!!");
}
```



// Evaluates to false so "It's Lunchtime Already" is logged

```
if (!isLunchTime) {
  console.log("Not Lunchtime Already!!");
} else {
  console.log("It's Lunchtime Already !!");
}
```

// Else if allows you to test more than one condition

// The first condition is false, so the second condition is evaluated.

Logs "Cost Rating: \$\$"

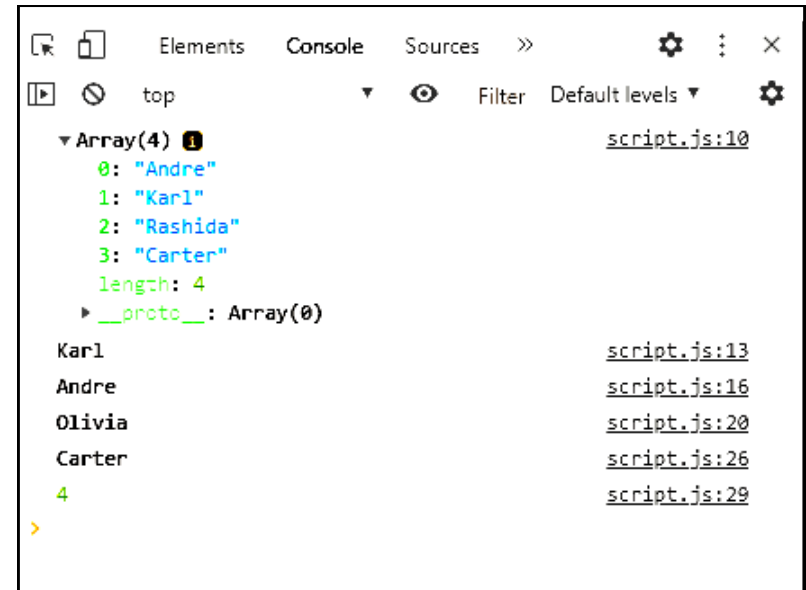
```
if (lunchBill < 10) {
  console.log("Cost Rating: $");
} else if (lunchBill >= 10 && lunchBill < 15) {
  console.log("Cost Rating: $$");
} else {
  console.log("Cost Rating: $$$");
}
```

Arrays

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Arrays</title>
</head>
<body>

  <h1 style="text-align:center">Open the Console to See the Magic </h1>
  <script src="script.js"></script>
</body>
</html>
```

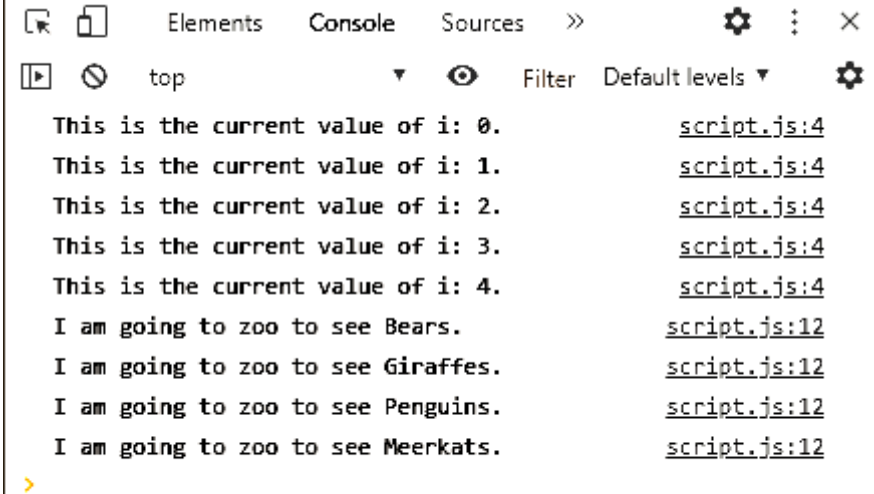
```
// So far, we have been storing one piece of data in variables
var name = "Andre";
var pets = 3;
var isStudent = true;
// To store groups of data in a single variable, we use arrays
var names = ["Andre", "Karl", "Rashida", "Olivia"];
// The entire array can be accessed by using the array's name
console.log(names);
// To log a single element, we use the name of the array with the index in brackets
console.log(names[1]);
// Arrays are zero-indexed, so the index of first element in the array is 0
console.log(names[0]);
// We can also use index to replace data in an array
// Returns "Olivia"
console.log(names[3]);
//Replaces "Olivia" with "Carter"
names[3] = "Carter";
// Logs "Carter"
console.log(names[3]);
// We use the array's length property to determine how many elements are in the array
console.log(names.length);
```



Iterations

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Iteration</title>
</head>
<body>

  <h1 style="text-align:center">Open the Console to See the Magic ☐</h1>
  <script src="script.js"></script>
</body>
</html>
```



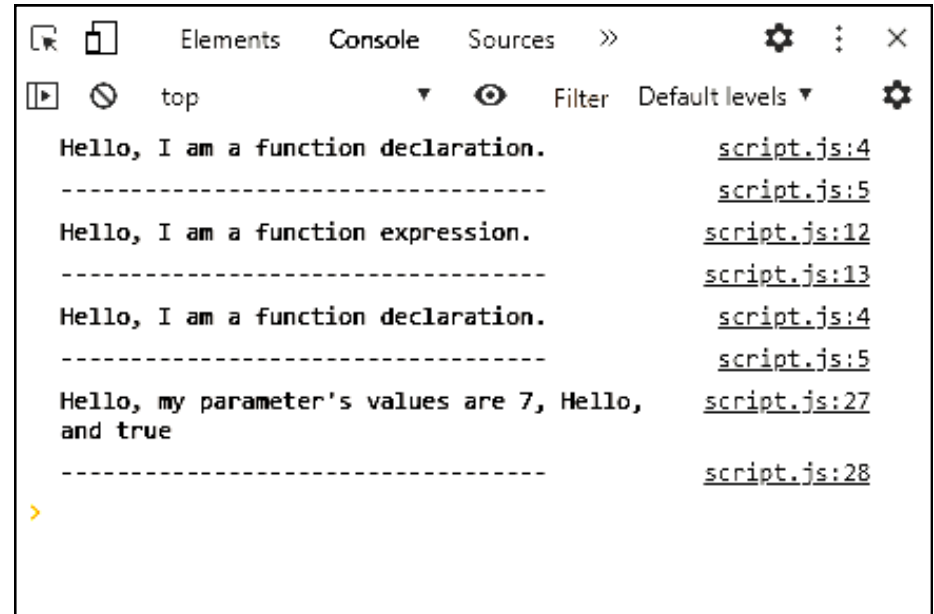
```
This is the current value of i: 0.      script.js:4
This is the current value of i: 1.      script.js:4
This is the current value of i: 2.      script.js:4
This is the current value of i: 3.      script.js:4
This is the current value of i: 4.      script.js:4
I am going to zoo to see Bears.         script.js:12
I am going to zoo to see Giraffes.      script.js:12
I am going to zoo to see Penguins.      script.js:12
I am going to zoo to see Meerkats.      script.js:12
```

```
// We use a for-loop to execute code more than once
for (var i = 0; i < 5; i++) {
  // This is the block of code that will run each time
  console.log("This is the current value of i: " + i + ".");
}
// For-loops are often used to iterate over arrays
var zooAnimals = ["Bears", "Giraffes", "Penguins", "Meerkats"];
//To determine how many times the loop should execute, we use the array's length
for (var i = 0; i < zooAnimals.length; i++) {
  console.log("I am going to zoo to see " + zooAnimals[i] + ".");
}
```

Functions

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Functions</title>
</head>
<body>

  <h1 style="text-align:center">Open the Console to See the Magic </h1>
  <script src="script.js"></script>
</body>
</html>
```



// Functions are reusable blocks of code that perform a specific task

// This is a function declaration

```
function declareHello() {
  console.log("Hello, I am a function declaration.");
  console.log("-----");
  // Return stops the execution of a function
  return;
}
```

// This is a function expression

```
var expressHello = function() {
  console.log("Hello, I am a function expression.");
  console.log("-----");
  return;
};
```

// Functions must be called to execute

```
declareHello();
expressHello();
```

//Functions can be called again to make the block of code execute again

declareHello();

// Functions can take parameters.

// Parameters give a name to the data to be passed into the function

```
function declareHelloAgain(x,y,z) {
  console.log("Hello, my parameter's values are " + x + ", " + y + ", and " + z);
  console.log("-----");
  return;
}
```

// Function arguments give parameters their values

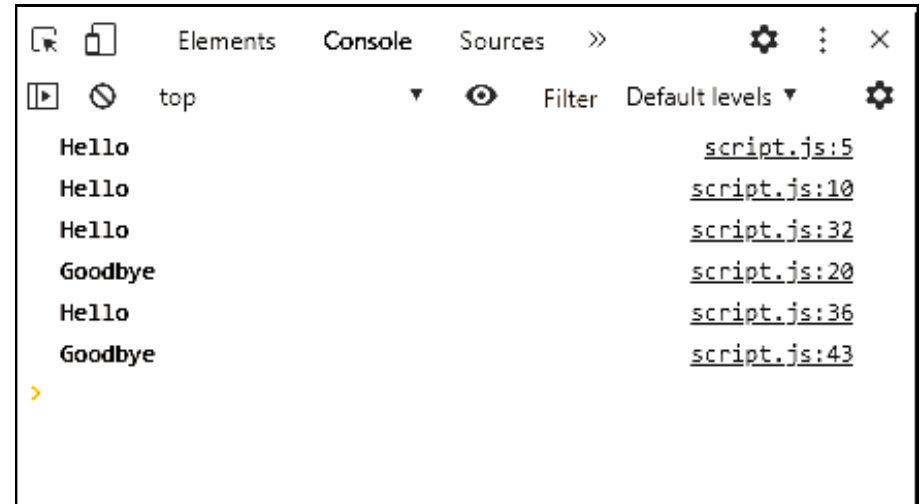
// Here the parameter x is given the value 7 when the function is called

declareHelloAgain(7, "Hello", true);

Scope

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Scope</title>
</head>
<body>

  <h1 style="text-align:center">Open the Console to See the Magic ☐</h1>
  <script src="script.js"></script>
</body>
</html>
```



// A variable declared in global scope is available to all functions

```
var hello = "Hello";
function sayHello() {
  console.log(hello);
  return;
}
var sayHelloAgain = function () {
  console.log(hello);
  return;
};
sayHello();
sayHelloAgain();
```

//A variable declared in local scope is only available to that function

```
function sayGoodbye() {
  var goodbye = "Goodbye";
  console.log(goodbye);
  return;
}
```

//This will throw an error

```
var sayGoodbyeAgain = function () {
  console.log(goodbye);
  return;
};
```

// Shadowing happens when the same variable is used in the local and global scope

```
var shadow = "Hello";
console.log(shadow);
```

// Logs "Hello"

```
function sayWhat() {
  console.log(shadow);
  return;
}
```

//Logs "Goodbye"

```
var sayWhatAgain = function () {
  var shadow = "Goodbye";
  console.log(shadow);
};
sayGoodbye();
sayWhat();
sayWhatAgain("Hello", true);
```

Methods

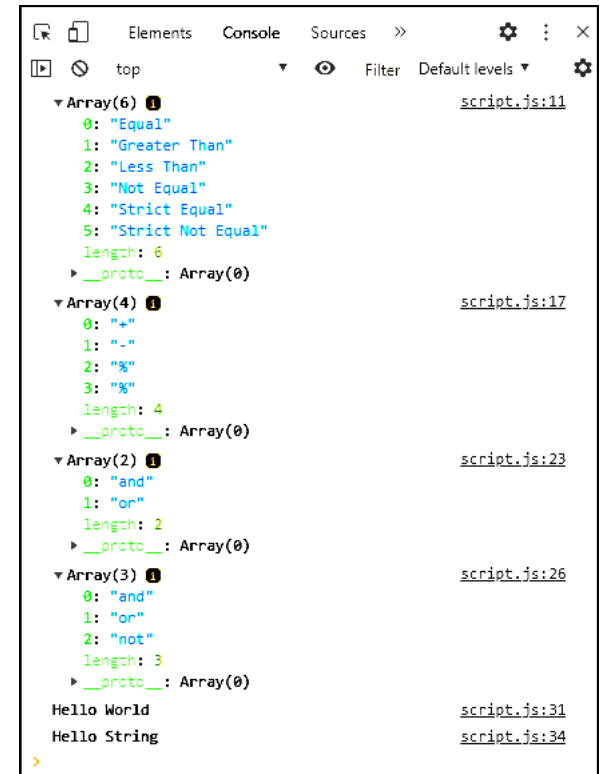
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Methods</title>
</head>
<body>

  <h1 style="text-align:center">Open the Console to See the Magic ☐</h1>
  <script src="script.js"></script>
</body>
</html>
```

```
var comparisonOperators = ["Equal", "Not Equal", "Strict Equal",
    "Strict Not Equal", "Greater Than", "Less Than"];
var arithmeticOperators = ["+", "-", "%"];
var logicalOperators = ["and", "or", "not"];
var myString = "Hello String";
```

//Array Methods

```
// Sorts comparisonOperators array and returns the sorted array
comparisonOperators.sort();
//Logs sorted array
console.log(comparisonOperators);
// Adds elements to end of an array. Takes in at least one parameter
arithmeticOperators.push("%");
//Logs array with element "%" added to end
console.log(arithmeticOperators);
//Returns selected elements as a new array.
var logicalOperatorsSliced = logicalOperators.slice(0,2);
//Logs new array
console.log(logicalOperatorsSliced);
// The original array is unchanged
console.log(logicalOperators);
```



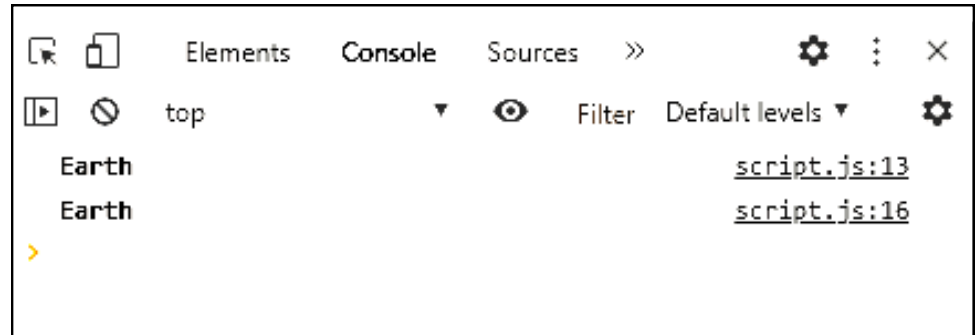
//String Methods

```
//Replaces "String" with "World" and returns new string
var myNewString = myString.replace("String", "World");
console.log(myNewString);
//The original string is unchanged
console.log(myString);
```

Objects

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Objects</title>
</head>
<body>

  <h1 style="text-align:center">Open the Console to See the Magic ☐</h1>
  <script src="script.js"></script>
</body>
</html>
```



// Objects are a collection of properties

```
var planet = {
```

// Properties are made up of key-value pairs

```
  name: "Earth",
```

```
  age: "4.543 billion years",
```

```
  moons: 1,
```

```
  isPopulated: true,
```

```
  population: "7.594 billion"
```

```
};
```

// To access a property's value that is a string, number or boolean,

use the object's name and the associated key

// Uses dot notation and logs "Earth"

```
console.log(planet.name);
```

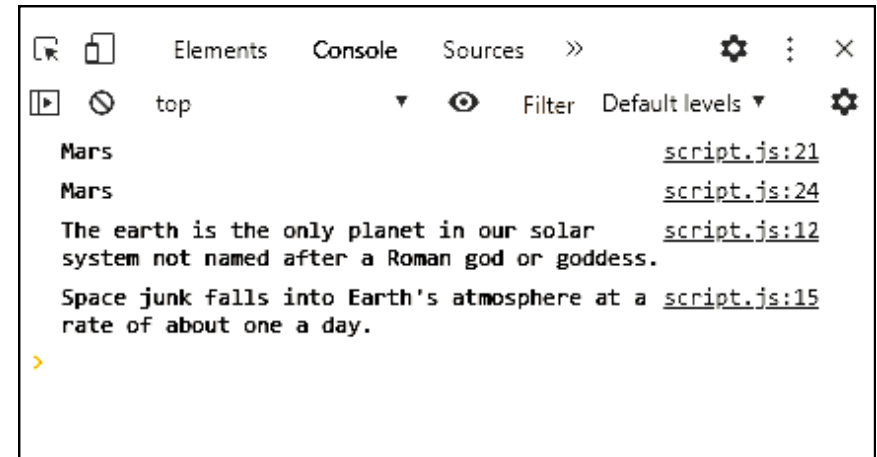
// Uses bracket notation and logs "Earth"

```
console.log(planet["name"]);
```

Object-Methods

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Objects</title>
</head>
<body>

  <h1 style="text-align:center">Open the Console to See the Magic ☐</h1>
  <script src="script.js"></script>
</body>
</html>
```



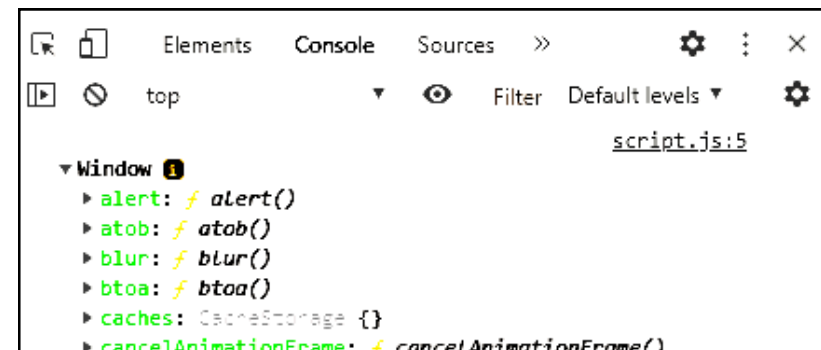
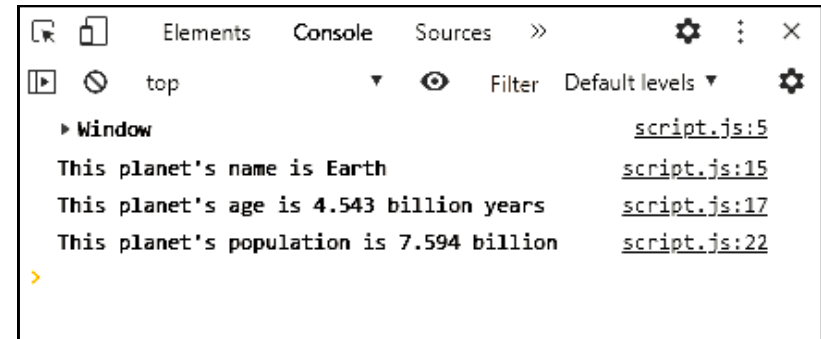
```
// Objects can store more than primitive data types like strings, booleans and numbers
var planet = {
  name: "Earth",
  age: "4.543 billion years",
  moons: 1,
  isPopulated: true,
  population: "7.594 billion",
  // Objects can store arrays in a key-value pair
  neighboringPlanets: ["Mars", "Venus"],
  // Objects can also store methods
  tellFunFact: function () {
    console.log("The earth is the only planet in our solar system not named after a Roman god or goddess.");
  },
  showWarning: function () {
    console.log("Space junk falls into Earth's atmosphere at a rate of about one a day.");
  }
};

// To access a value in an array, use the name of the object, the key and the index.
// Logs "Mars" using dot notation
console.log(planet.neighboringPlanets[0]);
// Logs "Mars" using bracket notation
console.log(planet["neighboringPlanets"][0]);
// To call a method, use the name of the object and the key. Don't forget the ()!
planet.tellFunFact();
planet.showWarning();
```

Object This

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Objects</title>
</head>
<body>

  <h1 style="text-align:center">Open the Console to See the Magic ☐</h1>
  <script src="script.js"></script>
</body>
</html>
```



```
// The default keyword "this" refers to the global object
// In a browser, the global object is the Window
// Logs Window
console.log(this);
// When the keyword "this" is used inside of an object like planet,
// "this" belongs to that object
var planet = {
  name: "Earth",
  age: "4.543 billion years",
  isPopulated: true,
  population: "7.594 billion",
  logFacts: function () {
    //Logs "This planet's name is Earth"
    console.log("This planet's name is " + this.name);
    //Logs "This planet's age is 4.543 billion years"
    console.log("This planet's age is " + this.age);
  },
  logPopulation: function () {
    if (this.isPopulated) {
      // Logs "This planet's population is 7.594 billion"
      console.log("This planet's population is " + this.population);
    } else {
      console.log("The planet is unpopulated");
    }
  }
};
```

```
// Calls object methods
planet.logFacts();
planet.logPopulation();
```

Git Branch

Now use the `cd` command to navigate into the newly created directory. Initialize an empty Git repo in the directory with the following command:

```
cd git_branch_demo
git init
```

Open the `git_branch_demo` directory in VS Code.

Now that we're ready to create the first feature, we need to create a new branch. Remember, the goal is to avoid working in the main branch, so that we can make mistakes on the new feature without damaging code that already works.

```
touch README.md
git add -A
git commit -m "creating a top level readme"
```

The Git command for creating a new named branch in your repo is `git checkout -b feature/<feature-name>`. Pick a name that is associated with the feature that you will be working on. `feature/` reminds us that each branch is dedicated to a specific feature, while `<feature-name>` is the name of the feature. In this case, we'll be creating a JS file, so let's call it `create-js-file`, as follows:

```
git checkout -b feature/create-js-file
```

This command will create the new branch, and then checkout to it. So you should now be on the `feature/create-js-file` branch.

The Git command `git branch` allows us to see a list of existing branches. Run the following command so that we can confirm that the `feature/create-js-file` branch was created:

```
git branch
```

Important: We also have the option to create a branch and switch over to it at the same time by entering `git checkout -b <branch-name>`.

Finally, add and commit the changes that you made, as follows:

```
git add -A
git commit -m "Created index.js and added text to the file"
```

Now that the feature is complete, you can merge the feature branch with main. First you need to switch back to main from `feature/create-js-file`. Remember, it's always a good idea to confirm that you're on the correct branch using the `git branch` command. See the following example:

```
git checkout main
git branch
```

Important: Git won't let you switch to a different branch until you have added, committed, and pushed any changes that you made to your feature branch. If you try to switch without pushing your code, Git will send you a reminder to push the changes that you made before switching branches.

Once we're in main, notice that we no longer have a `index.js` file in the directory. What happened? main is currently behind `feature/create-js-file` and we still need to merge the feature branch with the main codebase. To merge, add the following code to the command line:

```
git merge feature/create-js-file
```

Now the directory should include the `index.js` file that we created, along with whatever text we added to the file.

We could potentially generate a huge list of feature branches while working on a large project. So to avoid confusion and stay organized, it is good practice to close a branch once a feature is completed and merged. Because we're finished with this feature and the code is now included in main, we no longer need the isolated environment of that branch. We can always open another branch to fix future problems. But for now, we can safely close the feature branch by issuing the following command:

```
git branch -d feature/create-js-file
```

Congratulations, you've now completed your first branch lifecycle! You created an isolated environment on a new branch so that you could write and test code for a new feature, the `index.js` file. Once you finished adding text to `index.js`, you merged the feature branch with the main codebase on main. You then closed the feature branch, because you no longer needed to work on the `index.js` file.

Hints

You'll come up with your own naming conventions for branches when you're working on your own project. Try to be descriptive but concise to help other developers (or your future self) understand what is happening in each branch.

Popular naming conventions in the field include `feature/<feature-name>`, `issue/<issue-reference>`, etc. So it is a good idea to practice these conventions while you're learning.