

Software Structure

By Pargol Poshtareh

Lecture 1

Outline

- What is JavaScript?
 - History
 - Uses
- Adding JavaScript to HTML
- JavaScript syntax
- JavaScript events
- JavaScript classes
- The HTML Document Object Model

What is JavaScript?

- JavaScript is a programming language for use in HTML pages
- Invented in 1995 at Netscape Corporation
- JavaScript has nothing to do with Java
- JavaScript programs are run by an interpreter built into the user's web browser

What can JavaScript Do?

- JavaScript can dynamically modify an HTML page
- JavaScript can react to user input
- JavaScript can validate user input
- JavaScript can be used to create cookies (yum!)
- JavaScript is a full-featured programming language
- JavaScript user interaction *does not require any communication with the server*

Using JavaScript in your HTML

- JavaScript can be inserted into documents by using the SCRIPT tag

```
<html>
<head>
<title>Hello World in JavaScript</title>
</head>
<body>
  <script type="text/javascript">
    document.write("Hello World!");
  </script>
</body>
</html>
```

Where to Put your Scripts

- You can have any number of scripts
- Scripts can be placed in the HEAD or in the BODY
 - In the HEAD, scripts are run before the page is displayed
 - In the BODY, scripts are run as the page is displayed
- In the HEAD is the right place to define functions and variables that are used by scripts within the BODY

Using JavaScript in your HTML

```
<html>
<head>
<title>Hello World in JavaScript</title>
<script type="text/javascript">
  function helloWorld() {
    document.write("Hello World!");
  }
</script>
</head>
<body>
  <script type="text/javascript">
    helloWorld();
  </script>
</body>
</html>
```

External Scripts

- Scripts can also be loaded from an external file
- This is useful if you have a complicated script or set of subroutines that are used in several different documents

```
<script src="myscript.js"></script>
```


Exercise:

Implement the java script “hello my first JS code” in three way that you learn.

Data Type and Variables:

1) Numbers

Defining a number in JavaScript is actually pretty simple. The **Number** data type includes any positive or negative integer, as well as decimals. Entering a number into the console will return it right back to you.

Arithmetic operations:

You can also perform calculations with numbers pretty easily. Basically type out an expression the way you would type it in a calculator.

ex : $3 + 2.1$

Return: 5.1

QUESTION:

Enter the expressions (one at a time) into the console and determine what each expression evaluates to.

a. $2 + 10 - 19 + 4 - 90 + 1$

b. $-20 + -19 - (-10) - (-1) + 24$

c. $(10/5) * 4 - 20$

d. $4096 \% 12$

Comparing numbers

What about comparing numbers? Can you do that? Well of course you can!

Just like in mathematics, you can compare two numbers to see if one's greater than, less than, or equal to the other.

5 > 10 Returns: false

5 < 10 Returns: true

5 == 10 Returns: false

Comparisons between numbers will either evaluate to true or false. Here are some more examples, so you can try it out!

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or Equal to
>=	Greater than or Equal to
==	Equal to
!=	Not Equal to

QUESTION

Enter the expressions (one at a time) into the console and determine what each expression evaluates to.

a) $43 > 47$

b) $12 == 17$

c) $3 \leq 3$

d) $1 \neq 0$

2) Comments

```
// this is a single-line comment
```

```
/*  
this is  
a multi-line  
comment  
*/
```

Excercise:

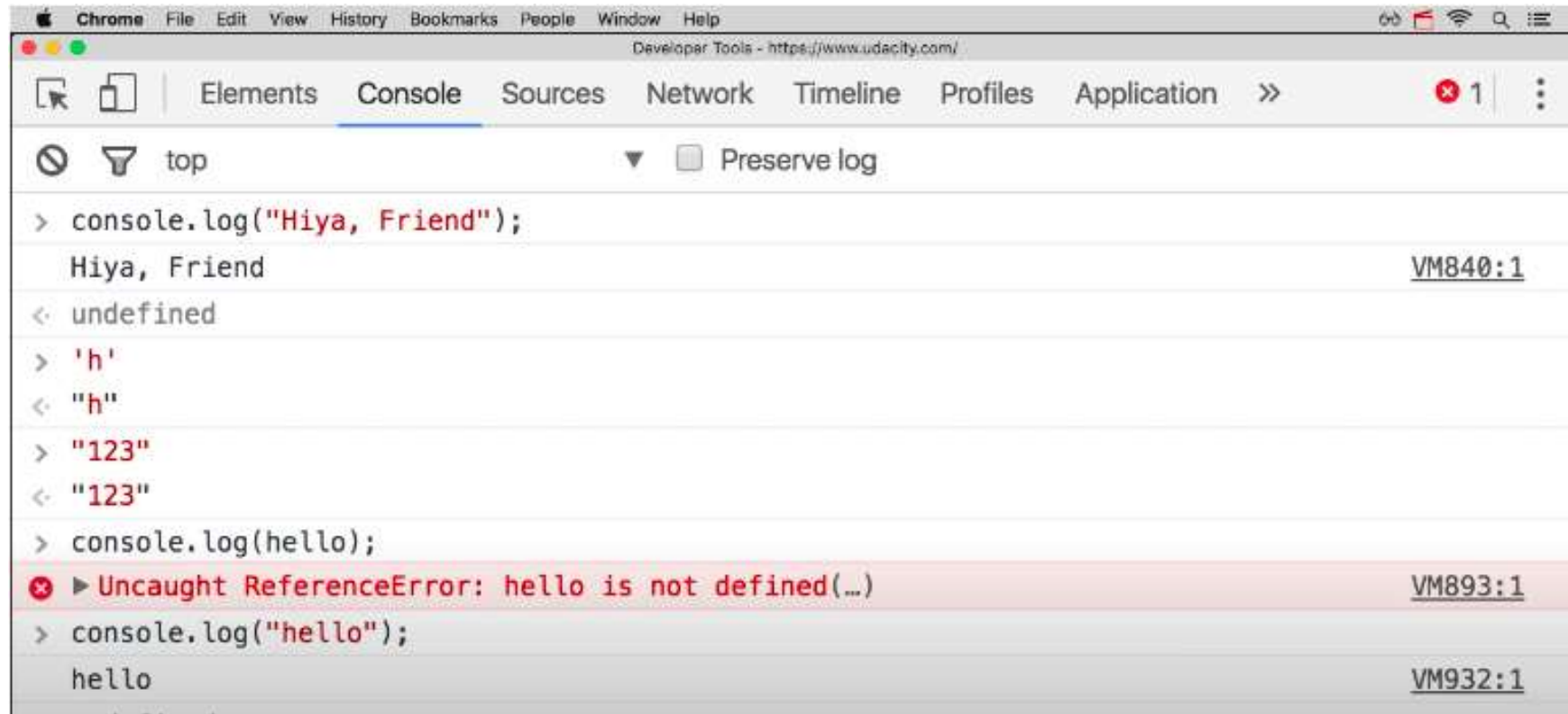
Write an expression that uses at least 3 different arithmetic operators.

The expression should equal 42.

Hint: +, -, *, /, and % are possible arithmetic operators

```
1- /*
2  * Programming Quiz: First Expression (2-1)
3  * Write an expression that uses at least three, distinct, arithmetic operators
4  * to log the number 42 to the console.
5  */
6
7- /*
8  * QUIZ REQUIREMENTS
9  * 1. Your code should print the value `42`
10 * 2. You should use at least 3 distinct operators. (`+`, `-`, `*`, `/`, or `%`)
11 * 3. Your code should not be empty
12 */
13
14
15 // this expression equals 4, change it to equal 42
16 console.log(
```


3) String



The screenshot shows the Chrome Developer Tools Console with the following content:

```
Chrome File Edit View History Bookmarks People Window Help
Developer Tools - https://www.udacity.com/
Elements Console Sources Network Timeline Profiles Application >>
top [x] Preserve log
> console.log("Hiya, Friend");
Hiya, Friend VM840:1
< undefined
> 'h'
< "h"
> "123"
< "123"
> console.log(hello);
✖ ▶ Uncaught ReferenceError: hello is not defined(...) VM893:1
> console.log("hello");
hello VM932:1
```

The console shows a series of JavaScript commands and their outputs. The first command, `console.log("Hiya, Friend");`, outputs `Hiya, Friend`. The second command, `< undefined`, shows the return value of the previous log statement. The third command, `> 'h'`, outputs `< "h"`. The fourth command, `> "123"`, outputs `< "123"`. The fifth command, `> console.log(hello);`, results in an `Uncaught ReferenceError: hello is not defined(...)` error. The sixth command, `> console.log("hello");`, outputs `hello`.

String concatenation

Strings are a collection of characters enclosed inside double or single quotes. You can use strings to represent data like sentences, names, addresses, and more. Did you know you can even add strings together? In JavaScript, this is called **concatenating**. Concatenating two strings together is actually pretty simple!

"Hello," + " New York City"

Returns: "Hello, New York City"

You will see other ways to concatenate and do even more with strings later in this course.

But for now, practice using the addition + operator.

QUESTION :

What's the result with "hello" + "world"?

What do you think will happen when you type "Hello + 5*10" into the JavaScript console?

What do you think will happen when you type "Hello" + 5*10 into the console?

3) Variable

With variables, you no longer need to work with one-time-use data.

At the beginning of this course, you declared the value of a string, but you didn't have a way to access or reuse the string later.

```
"Hello"; // Here's a String "Hello"
```

```
"Hello" + " World"; // Here's a new String (also with the value "Hello") concatenated with " World"
```

Storing the value of a string in a variable is like packing it away for later use.

```
var greeting = "Hello";
```

Now, if you want to use "Hello" in a variety of sentences, you don't need to duplicate "Hello" strings. You can just reuse the greeting variable.

```
greeting + " World!";
```

Returns: Hello World!

```
greeting + " Mike!";
```

Returns: Hello Mike!

You can also change the start of the greeting by *reassigning* a new string value to the variable greeting.

```
greeting = "Hola"; greeting + " World!";
```

Returns: Hola World!

```
greeting + " Mike!";
```

Returns: Hola Mike!

Naming conventions

When you create a variable, you write the name of the variable using camelCase (the first word is lowercase, and all following words are uppercase). Also try to use a variable name that accurately, but succinctly describes what the data is about.

```
var totalAfterTax = 53.03; // uses camelCase if the variable name is multiple words
```

```
var tip = 8; // uses lowercase if the variable name is one word
```

Not using camelCase for your variables names is not going to necessarily *break* anything in JavaScript. But there are recommended style guides used in all programming languages that help keep code consistent, clean, and easy-to-read. This is especially important when working on larger projects that will be accessed by multiple developers.

QUESTION :

Which of these are good variable names?

- `var thingy = 1;`
- `var count = 1;`
- `var postLiked = false;`
- `var firstname = "Richard";`

Exercise:

To convert Celsius to Fahrenheit, you can use the following formula:

$$F = C \times 1.8 + 32$$

Directions:

Use this equation and the variables Fahrenheit and Celsius to print the Fahrenheit equivalent of 12°C.

```
/*The Celsius-to-Fahrenheit formula:
 *
 *   F = C x 1.8 + 32
 *
 * 1. Set the fahrenheit variable to the correct value using
 *    the celsius variable and the formula above
 * 2. Log the fahrenheit variable to the console
 *
 */

/*
 * QUIZ REQUIREMENTS
 * 1. Your code should have a variable `celsius`
 * 2. Your code should have a variable `fahrenheit`
 * 3. Your variable `celsius` should equal `12`
 * 4. Your variable `fahrenheit` should produce the output equal `53.6`
 * 5. Your variable `fahrenheit` declaration should use the `celsius` variable
 * 6. Your variable `fahrenheit` should have the correct formula
 * 7. Your code should log the `fahrenheit` variable
 * 8. Your code should not be empty
 */

var celsius = 12;
var fahrenheit = /* convert celsius to fahrenheit here */

console.log(/* print out result here */);
```


4) Indexing

Did you know that you can access individual characters in a string? To access an individual character, you can use the character's location in the string, called its **index**.

Just put the index of the character inside square brackets (starting with [0] as the first character) immediately after the string. For example:

```
"James"[0];
```

Returns: "J"

```
var name = "James";  
name[0];
```

Returns: "J"

Question :

What character will be printed to the JavaScript console after running the following lines of code.

```
var quote = "Stay awhile and listen!"; console.log(quote[6]);
```

Escaping strings

There are some cases where you might want to create a string that contains more than just numbers and letters. For example, what if you want to use quotes in a string?

"The man whispered, "please speak to me."" Uncaught SyntaxError:

Unexpected identifier

If you try to use quotes within a string, you will receive a `SyntaxError` like the one above.

If you want to use quotes *inside a string*, and have JavaScript not misunderstand your intentions, you'll need a different way to write quotes. Thankfully, JavaScript has a way to do this using the backslash character (`\`).

"The man whispered, \"please speak to me.\""

Returns: The man whispered, "please speak to me."

Special characters

Quotes aren't the only **special characters** that need to be escaped, there's actually quite a few. However, to keep it simple, here's a list of some common special characters in JavaScript.

"Up up\n\tdown down"

Returns:

Up up
down down

Code	Character
\\	\ (backslash)
\"	" (double quote)
\'	' (single quote)
\n	newline
\t	tab

Comparing strings

Another way to work with strings is by comparing them. You've seen the comparison operators `==` and `!=` when you compared numbers for equality. You can also use them with strings! For example, let's compare the string "Yes" to "yes".

```
"Yes" == "yes"
```

Returns: false

```
'Y' != 'y'
```

Returns: true

Exercise :

Build a single string that resembles the following joke.

Why couldn't the shoes go out and play?

They were all "tied" up!

```
/*  
 * Programming Quiz: All Tied Up (2-5)  
  
 * Quiz Requirements  
 * 1. Your code should have a variable joke  
 * 2. Your joke should use only one string  
 * 3. Your joke should match the expected format  
 */  
var joke = /* write your joke here */  
console.log(joke);
```

5) Booleans

A Boolean variable can take either of two values - true or false. For example,

```
var studentName = "John";
```

```
var haveEnrolledInCourse = true;
```

```
var haveCompletedTheCourse = false;
```

A Boolean variable is mainly essential in evaluating the outcome of conditionals (comparisons).

The result of a comparison is always a Boolean variable.

We'll study conditionals in our upcoming lesson, but let's look at our previous example to understand the role of Boolean in conditional:

```
if (haveEnrolledInCourse){ console.log("Welcome "+studentName+" to School!"); // Will  
run only if haveEnrolledInCourse is true }
```

In general cases (regular equality check), a **true** corresponds to number 1, whereas **false** represents a number 0. For example:

What is NaN?

NaN stands for "Not-A-Number" and it's often returned indicating an error with number operations. For instance, if you wrote some code that performed a math calculation, and the calculation failed to produce a valid number, NaN might be returned.

// calculating the square root of a negative number will return NaN

```
Math.sqrt(-10)
```

// trying to divide a string by 5 will return NaN

```
"hello"/5
```

Equality

So far, you've seen how you can use `==` and `!=` to compare numbers and strings for equality. However, if you use `==` and `!=` in situations where the values that you're comparing have different data-types, it can lead to some interesting results. For

example,

`"1" == 1`

Returns: true

`0 == false`

Returns: true. The `==` operator is unable to differentiate 0 from false.

`' ' == false`

Returns: true. Both the operands on either side of the `==` operator are first converted to zero, before comparison.

Implicit type coercion

JavaScript is known as a *loosely typed language*.

Basically, this means that when you're writing JavaScript code, you do not need to specify data types. Instead, when your code is interpreted by the JavaScript engine it will automatically be converted into the "appropriate" data type. This is called *implicit type coercion* and you've already seen examples like this before when you tried to concatenate strings with numbers.

"julia" + 1

Returns: "julia1"

QUESTION:

What value do you think the result of "Hello" % 10 will be?

- 0
- "Hello10"
- 10
- SyntaxError
- NaN

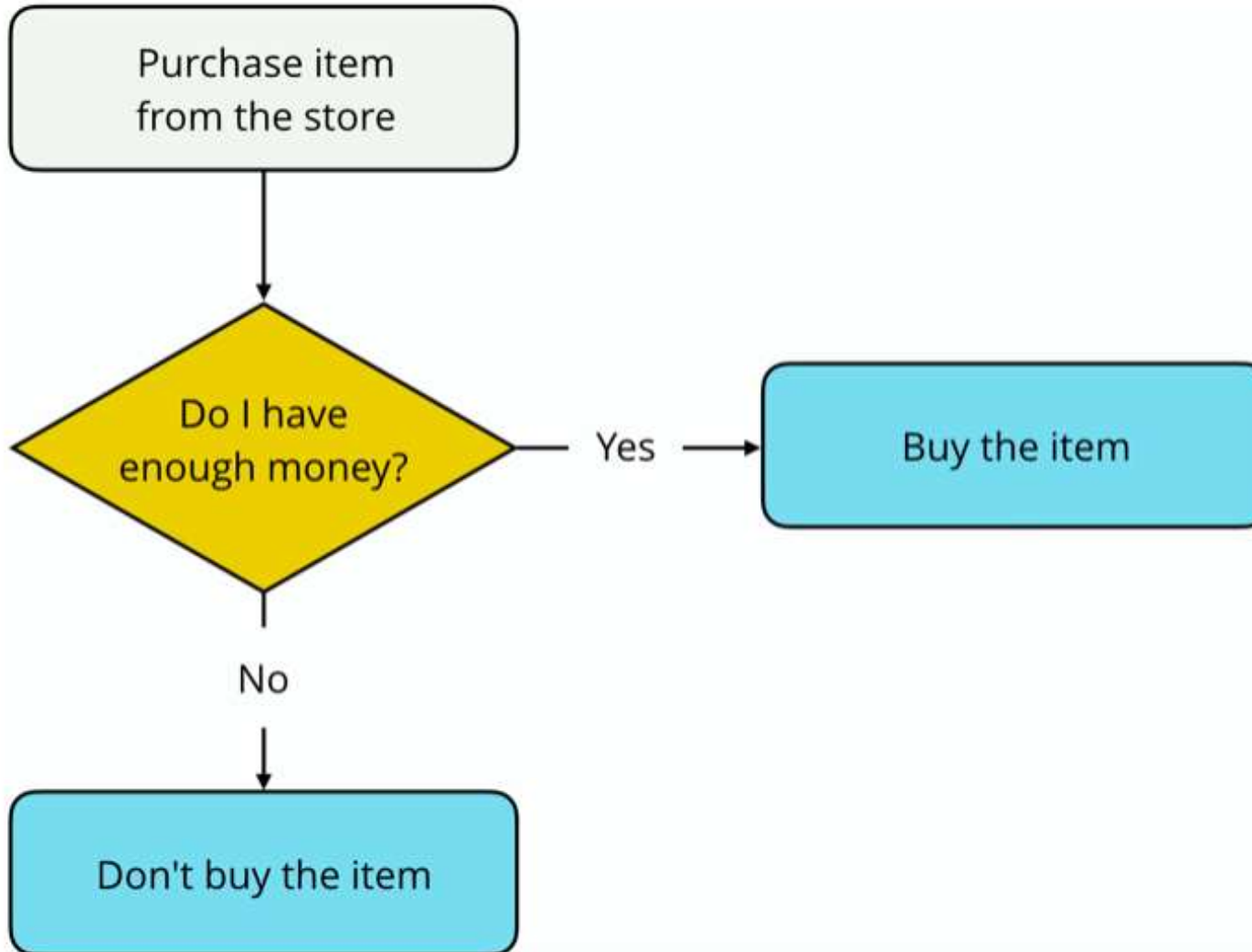
Exercise:

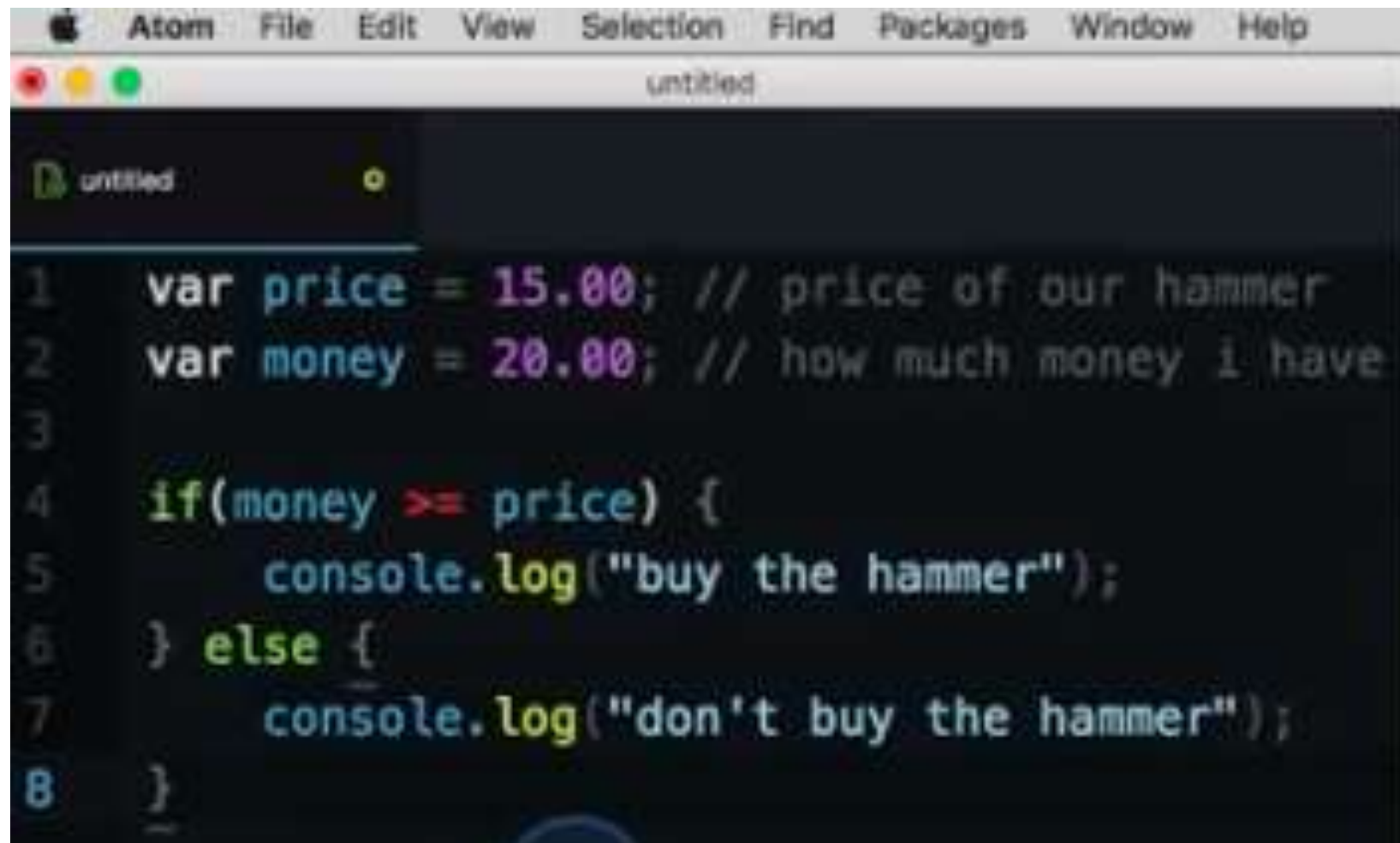
Create a variable called `bill` and assign it the result of $10.25 + 3.99 + 7.15$ (don't perform the calculation yourself, let JavaScript do it!). Next, create a variable called `tip` and assign it the result of multiplying `bill` by a 15% tip rate. Finally, add the `bill` and `tip` together and store it into a variable called `total`.

Print the `total` to the JavaScript console.

Hint: *15% in decimal form is written as 0.15_._*

Flowchart:



A screenshot of the Atom code editor interface. The window title is 'untitled'. The menu bar includes 'Atom', 'File', 'Edit', 'View', 'Selection', 'Find', 'Packages', 'Window', and 'Help'. The editor shows a JavaScript code snippet with line numbers 1 through 8 on the left. The code is: 1 var price = 15.00; // price of our hammer, 2 var money = 20.00; // how much money i have, 3, 4 if(money >= price) {, 5 console.log("buy the hammer");, 6 } else {, 7 console.log("don't buy the hammer");, 8 }

```
1  var price = 15.00; // price of our hammer
2  var money = 20.00; // how much money i have
3
4  if(money >= price) {
5      console.log("buy the hammer");
6  } else {
7      console.log("don't buy the hammer");
8  }
```

If...else statements

If...else statements allow you to execute certain pieces of code based on a condition, or set of conditions, being met.

```
if (/* this expression is true */) {  
    // run this code  
}  
else {  
    // run this code  
}
```

This is extremely helpful because it allows you to choose which piece of code you want to run based on the result of an expression.

Example,

```
var a = 1;  
var b = 2;  
  
if (a > b) {  
    console.log("a is greater than b");  
} else {  
    console.log("a is less than or equal to b");  
}
```

Prints: "a is less than or equal to b"

Else if statements

In JavaScript, you can represent this secondary check by using an extra if statement called an **else if statement**.

```
var weather = "sunny";

if (weather === "snow") {
  console.log("Bring a coat.");
} else if (weather === "rain") {
  console.log("Bring a rain jacket.");
} else {
  console.log("Wear what you have on.")
;
}
```

***Prints:** Wear what you have on.*

QUESTION :

What will be printed to the console if the following code is run?

```
var runner = "Kendyll";
var position = 2;
var medal;

if(position === 1) {
  medal = "gold";
} else if(position === 2) {
  medal = "silver";
} else if(position === 3) {
  medal = "bronze";
} else {
  medal = "pat on the back";
}

console.log(runner + " received a " + medal + " medal.");
```

Exercise:

Write a series of conditional statements that:

- Prints "not a group" if musicians is less than or equal to 0
- Prints "solo" if musicians is equal to 1
- Prints "duet" if musicians is equal to 2
- Prints "trio" if musicians is equal to 3
- Prints "quartet" if musicians is equal to 4
- Prints "this is a large group" if musicians is greater than 4

Logical expressions

Logical expressions are similar to mathematical expressions, except logical expressions evaluate to either *true* or *false*.

`11 != 12`

Returns: true

Similar to mathematical expressions that use +, -, *, / and %, there are logical operators `&&`, `||` and `!` that you can use to create more complex logical expressions.

Logical operators

Logical operators can be used in conjunction with boolean values (true and false) to create complex logical expressions.

By combining two boolean values together with a logical operator, you create a *logical expression* that returns another boolean value. Here's a table describing the different logical operators:

Operator	Meaning	Example	How it works
&&	Logical AND	value1 && value2	Returns <code>true</code> if both <code>value1</code> and <code>value2</code> evaluate to <code>true</code> .
	Logical OR	value1 value2	Returns <code>true</code> if either <code>value1</code> or <code>value2</code> (or even both!) evaluates to <code>true</code> .
!	Logical NOT	!value1	Returns the opposite of <code>value1</code> . If <code>value1</code> is <code>true</code> , then <code>!value1</code> is <code>false</code> .

&& (AND)

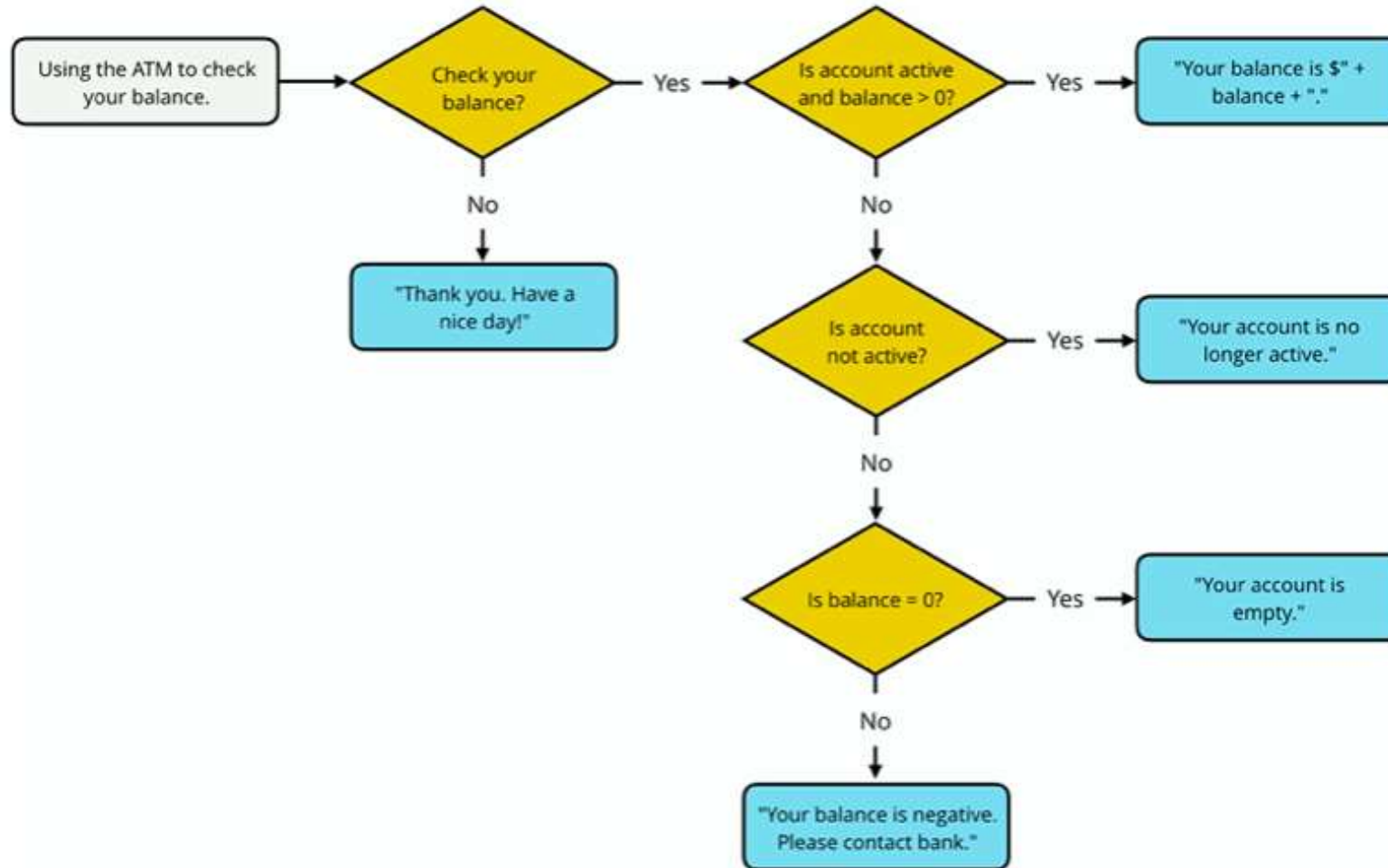
A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

|| (OR)

A	B	A B
true	true	true
true	false	true
false	true	true
false	false	false

Exercise:

Using the flowchart below, write the code to represent checking your balance at the ATM. The yellow diamonds represent conditional statements and the blue rectangles with rounded corners represent what should be printed to the console.



Flowchart for checking your balance at the ATM (Click the image to enlarge the flowchart).

switch statement

If you find yourself repeating **else if** statements in your code, where each condition is based on the same value, then it might be time to use a switch statement.

```
switch (option) {  
  case 1:  
    console.log("You selected option 1.");  
  case 2:  
    console.log("You selected option 2.");  
  case 3:  
    console.log("You selected option 3.");  
  case 4:  
    console.log("You selected option 4.");  
  case 5:  
    console.log("You selected option 5.");  
  case 6:  
    console.log("You selected option 6.");  
}
```

Break statement

The **break statement** can be used to terminate a switch statement and transfer control to the code following the terminated statement. By adding a break to each case clause, you fix the issue of the switch statement *falling-through* to other case clauses.

```
var option = 3;

switch (option) {
  case 1:
    console.log("You selected option 1.");
    break;
  case 2:
    console.log("You selected option 2.");
    break;
  case 3:
    console.log("You selected option 3.");
    break;
  case 4:
    console.log("You selected option 4.");
    break;
  case 5:
    console.log("You selected option 5.");
    break;
  case 6:
    console.log("You selected option 6.");
    break; // technically, not needed
}
```

While Loop

There are many different kinds of loops, but they all essentially do the same thing: they repeat an action some number of times.

Three main pieces of information that any loop should have are:

- 1.When to start:** The code that sets up the loop — defining the starting value of a variable for instance.
- 2.When to stop:** The logical condition to test whether the loop should continue.
- 3.How to get to the next item:** The incrementing or decrementing step — for example, $x = x * 3$ or $x = x - 1$

Here's a basic while loop example that includes all three parts.

```
var start = 0; // when to start
while (start < 10) { // when to stop
  console.log(start);
  start = start + 2; // how to get to the next item
}
```

Prints:

0
2
4
6
8

For Loop:

The **for loop** explicitly forces you to define the start point, stop point, and each step of the loop. In fact, you'll get an **Uncaught SyntaxError: Unexpected token)** if you leave out any of the three required pieces.

```
for ( start; stop; step ) {  
    // do this thing  
}
```

Here's an example of a for loop that prints out the values from 0 to 5. Notice the semicolons separating the different statements of the for loop: `var i = 0; i < 6; i = i + 1`

```
for (var i = 0; i < 6; i = i + 1) {  
    console.log("Printing out i = " + i);  
}
```

Prints:

Printing out i = 0

Printing out i = 1

Printing out i = 2

Printing out i = 3

Printing out i = 4

Printing out i = 5

Nested Loops:

Did you know you can also *nest* loops inside of each other? Paste this nested loop in your browser and take a look at what it prints out:

```
for (var x = 0; x < 5; x = x + 1) {  
  for (var y = 0; y < 3; y = y + 1) {  
    console.log(x + "," + y);  
  }  
}
```

Output:

0, 0
0, 1
0, 2
1, 0
1, 1
1, 2
2, 0
2, 1
2, 2
3, 0
3, 1
3, 2
4, 0
4, 1
4, 2

Here is a summary of operators you've learned so far:

```
x++ or ++x // same as x = x + 1  
x-- or --x // same as x = x - 1  
x += 3 // same as x = x + 3  
x -= 6 // same as x = x - 6  
x *= 2 // same as x = x * 2  
x /= 5 // same as x = x / 5
```

Exercise:

Rewrite the following **while** loop as a **for** loop:

```
var x = 9;  
while (x >= 1) {  
  console.log("hello " + x);  
  x = x - 1;  
}
```

Function

How to declare a function

Functions allow you to package up lines of code that you can use (and often reuse) in your programs.

The function can have parameter or not

```
function reverseString(reverseMe) {  
    // code to reverse a string!  
}
```

```
// accepts no parameters! parentheses are empty  
function sayHello() {  
    var message = "Hello!"  
    console.log(message);  
}
```


Return statements

In the sayHello() function above, a value is **printed** to the console with console.log, but not explicitly returned with a **return statement**. You can write a return statement by using the return keyword followed by the expression or value that you want to return.

```
// declares the sayHello function  
function sayHello() {  
  var message = "Hello!"  
  return message; // returns value instead of printing it  
}
```

How to *run* a function

Now, to get your function to *do something*, you have to **invoke** or **call** the function using the function name, followed by parentheses with any **arguments** that are passed into it. Functions are like machines. You can build the machine, but it won't do anything unless you also turn it on. Here's how you would call the `sayHello()` function from before, and then use the return value to print to the

```
// declares the sayHello function  
function sayHello() {  
    var message = "Hello!"  
    return message; // returns value instead of printing it  
}  
  
// function returns "Hello!" and console.log prints the return value  
console.log(sayHello());
```

Print :
"Hello!"

Parameters vs. Arguments

At first, it can be a bit tricky to know when something is either a parameter or an argument. The key difference is in where they show up in the code. A **parameter** is always going to be a *variable* name and appears in the function declaration. On the other hand, an **argument** is always going to be a *value* (i.e. any of the JavaScript data types - a number, a string, a boolean, etc.) and will always appear in the code when the function is called or invoked.

QUESTION

Use the following function to answer this question.

```
function findAverage(x, y) {  
  var answer = (x + y) / 2;  
  return answer;  
}  
  
var avg = findAverage(5, 9);
```

What value will be stored in the variable avg?

Exercise:

Write a function called `laugh()` that takes one parameter, `num` that represents the number of "ha"s to return.

TIP: You might need a loop to solve this!

Here's an example of the output and how to call the function that you will write:

`console.log(laugh(3));` Prints: "hahaha!"

Return Value :

```
function isThisWorking(input) {  
    console.log("Printing: isThisWorking was called and " + input + " was passed  
                in as an argument.");  
    return "Returning: I am returning this string!"; }  
isThisWorking(3);
```

***Prints:** "Printing: isThisWorking was called and 3 was passed in as an argument"*

***Returns:** "Returning: I am returning this string!"*

```
function isThisWorking(input) {  
    console.log("Printing: isThisWorking was called and " + input + " was passed  
                in as an argument.");  
}  
isThisWorking(3);
```

***Prints:** "Printing: isThisWorking was called and 3 was passed in as an argument"*

***Returns:** undefined*

QUESTION

What number will be "printed" (to the JavaScript console)?

```
function square(x) {  
    return x * x;  
}  
  
function subtractFour(x) {  
    return square(x) - 4;  
}  
  
console.log(subtractFour(5));
```

Using Return Values:

A function's return value can be stored in a variable or reused throughout your program as a function argument. Here, we have a function that adds two numbers together, and another function that divides a number by 2. We can find the average of 5 and 7 by using the `add()` function to add a pair of numbers together, and then by passing the sum of the two numbers `add(5, 7)` into the function `divideByTwo()` as an argument.

And finally, we can even store the final answer in a variable called `average` and use the variable to perform even more calculations in more places!

Next slide :

// returns the sum of two numbers

```
function add(x, y) {  
    return x + y;  
}
```

// returns the value of a number divided by 2

```
function divideByTwo(num) {  
    return num / 2;  
}
```

```
var sum = add(5, 7); // call the "add" function and store
```

```
var average = divideByTwo(sum); // call the "divideByTwo"
```

Scope:

global scope

identifiers can be
accessed everywhere
within your program

function scope

identifiers can be
accessed everywhere
inside the function it
was defined in

Global scope:

```
1 var james = "I'm looking for this book...";  
2  
3 function library() {  
4     var librarian = "Oh, you'll want to look in the classic  
5     literature section, follow me!";  
6     function classicLiterature() {  
7         var book = "Great Expectations";  
8     }  
9 }
```

Function scope:

```
1 var james = "I'm looking for this book...";  
2  
3 function library() {  
4     var librarian = "Oh, you'll want to look in the classic  
5     literature section, follow me!";  
6     function classicLiterature() {  
7         var book = "Great Expectations";  
8     }  
9 }
```

QUESTION 1 OF 2

Which of these variables a, b, c, or d, is defined in the global scope?

```
var a = 1;
function x() {
  var b = 2;
  function y() {
    var c = 3;
    function z() {
      var d = 4;
    }
    z();
  }
  y();
}

x();
```

QUESTION 2 OF 2

Where can you print out the value of variable c without resulting in an error?

```
var a = 1;
function x() {
  var b = 2;
  function y() {
    var c = 3;
    function z() {
      var d = 4;
    }
    z();
  }
  y();
}

x();
```

Shadowing or overwriting:

```
1  var bookTitle = "Le Petit Prince";
2  console.log(bookTitle);
3
4  function displayBookEnglish() {
5      bookTitle = "The Little Prince";
6      console.log(bookTitle);
7  }
8
9  displayBookEnglish();
10 console.log(bookTitle);
```

```
> "Le Petit Prince"
> "The Little Prince"
> "The Little Prince"
```

```
1  var bookTitle = "Le Petit Prince";
2  console.log(bookTitle);
3
4  function displayBookEnglish() {
5      var bookTitle = "The Little Prince";
6      console.log(bookTitle);
7  }
8
9  displayBookEnglish();
10 console.log(bookTitle);
```

```
> "Le Petit Prince"
> "The Little Prince"
> "Le Petit Prince"
```

Question :

Without pasting into your console, what do you think this code will print out?

```
var x = 1;

function addTwo() {
    x = x + 2;
}

addTwo();
x = x + 1;
console.log(x);
```


QUESTION

Without pasting into your console, what do you think this code will print out?

```
var x = 1;

function addTwo() {
  var x = x + 2;
}

addTwo();
x = x + 1;
console.log(x);
```

Using global variables

So you might be wondering:

"Why wouldn't I always use global variables? Then, I would never need to use function arguments since ALL my functions would have access to EVERYTHING!"

Well... Global variables might seem like a convenient idea at first, especially when you're writing small scripts and programs, but there are many reasons why you shouldn't use them unless you have to. For instance, global variables can conflict with other global variables of the same name. Once your programs get larger and larger, it'll get harder and harder to keep track and prevent this from happening.

There are also other reasons you'll learn more about in more advanced courses. But for now, just work on minimizing the use of global variables as much as possible.

What you've learned so far:

- If an identifier is declared in **global scope**, it's available *everywhere*.
- If an identifier is declared in **function scope**, it's available *in the function* it was declared in (even in functions declared inside the function).
- When trying to access an identifier, the JavaScript Engine will first look in the current function. If it doesn't find anything, it will continue to the next outer function to see if it can find the identifier there. It will keep doing this until it reaches the global scope.
- Global identifiers are a bad idea. They can lead to bad variable names, conflicting variable names, and messy code.

Hoisting

Sometimes your JavaScript code will produce errors that may seem counterintuitive at first. **Hoisting** is another one of those topics that might be the cause of some of these tricky errors you're debugging.

Let's take a look at an example:

```
1  findAverage(5, 9);  
2  
3  function findAverage(x, y) {  
4      var answer = (x + y) / 2;  
5      return answer;  
6  }
```

```
> 7
```

Exercise:

You're going to create a function called `buildTriangle()` that will accept an input (the triangle at its widest width) and will return the string representation of a triangle. See the example output below.

`buildTriar`

Returns:

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
```

Hint : We've given you one function `makeLine()` to start with. The function takes in a line length, and builds a line of asterisks and returns the line with a newline character.

```
function makeLine(length) {
  var line = "";
  for (var j = 1; j <= length; j++) {
    line += "* ";
  }
  return line + "\n";
}
```

Function Expressions

Once you know how to declare a function, a whole new set of possibilities will open up to you. For instance, remember how you can store anything you want in a variable? Well, in JavaScript, you can also store functions in variables. When a function is stored *inside* a variable it's called a **function expression**.

```
var catSays = function(max) {  
  var catMessage = "";  
  for (var i = 0; i < max; i++) {  
    catMessage += "meow ";  
  }  
  return catMessage;  
};
```

Notice how the `function` keyword no longer has a name.

```
var catSays = function(max) {  
    // code here  
};
```

It's an **anonymous function**, a function with no name, and you've stored it in a variable called `catSays`. And, if you try accessing the value of the variable `catSays`, you'll even see the function returned back to you.

`catSays;`
Returns:

```
function(max) {  
    var catMessage = ""  
    for (var i = 0; i < max; i++) {  
        catMessage += "meow ";  
    }  
    return catMessage;  
}
```

Functions as parameters:

Being able to store a function in a variable makes it really simple to pass the function into another function. A function that is passed into another function is called a **callback**. Let's say you had a `helloCat()` function, and you wanted it to return "Hello" followed by a string of "meows" like you had with `catSays`. Well, rather than redoing all of your hard work, you can make `helloCat()` accept a callback function, and pass in `catSays`.

Next....

// function expression catSays

```
var catSays = function(max) {  
  var catMessage = "";  
  for (var i = 0; i < max; i++) {  
    catMessage += "meow ";  
  }  
  return catMessage;  
};
```

// function declaration helloCat accepting a callback

```
function helloCat(callbackFunc) {  
  return "Hello " + callbackFunc(3);  
}
```

// pass in catSays as a callback function

```
helloCat(catSays);
```

Named function expressions:

Inline function expressions

A function expression is when a function is assigned to a variable. And, in JavaScript, this can also happen when you pass a function *inline* as an argument to another function. Take the favoriteMovie example for instance:

```
// Function expression that assigns the function displayFavorite
// to the variable favoriteMovie
var favoriteMovie = function displayFavorite(movieName) {
  console.log("My favorite movie is " + movieName);
};

// Function declaration that has two parameters: a function for displaying
// a message, along with a name of a movie
function movies(messageFunction, name) {
  messageFunction(name);
}

// Call the movies function, pass in the favoriteMovie function and name of movie
movies(favoriteMovie, "Finding Nemo");
```

Returns:
My favorite movie is Finding Nemo

But you could have bypassed the first assignment of the function, by passing the function to the `movies()` function inline.

```
// Function declaration that takes in two arguments: a function for displaying  
// a message, along with a name of a movie  
function movies(messageFunction, name) {  
  messageFunction(name);  
}  
  
// Call the movies function, pass in the function and name of movie  
movies(function displayFavorite(movieName) {  
  console.log("My favorite movie is " + movieName);  
}, "Finding Nemo");
```

Returns: *My favorite movie is Finding Nemo*

This type of syntax, writing function expressions that pass a function into another function inline, is really common in JavaScript. It can be a little tricky at first, but be patient, keep practicing, and you'll start to get the hang of it!

Why use anonymous inline function expressions?

Using an anonymous inline function expression might seem like a very not-useful thing at first. Why define a function that can only be used once and you can't even call it by name?

Anonymous inline function expressions are often used with function callbacks that are probably not going to be reused elsewhere. Yes, you could store the function in a variable, give it a name, and pass it in like you saw in the examples above. However, when you know the function is not going to be reused, it could save you many lines of code to just define it inline.

What you've learned so far:

Function Expression: When a function is assigned to a variable. The function can be named, or anonymous. Use the variable name to call a function defined in a function expression

```
// anonymous function expression  
var doSomething = function(y) {  
    return y + 1;  
};
```

```
// named function expression  
var doSomething = function addOne(y) {  
    return y + 1;  
};
```

```
// for either of the definitions above, call the function like this:  
doSomething(5);
```

Returns: 6

You can even pass a function into another function *inline*. This pattern is commonly used in JavaScript, and can be helpful streamlining your code.

```
// function declaration that takes in two arguments: a function for displaying  
// a message, along with a name of a movie  
function movies(messageFunction, name) {  
    messageFunction(name);  
}  
  
// call the movies function, pass in the function and name of movie  
movies(function displayFavorite(movieName) {  
    console.log("My favorite movie is " + movieName);  
}, "Finding Nemo");
```

Exercise:

Write an anonymous function expression that stores a function in a variable called "laugh" and outputs the number of "ha"s that you pass in as an argument.

```
laugh(3);
```

Returns: hahaha!

Exercise:

Directions:

Call the emotions() function so that it prints the output you see below, but instead of passing the laugh() function as an argument, pass an inline function expression instead.

```
emotions("happy", laugh(2));
```

Prints: "I am happy, haha!"

Arrays:

An **array** is useful because it stores multiple values into a single, organized data structure. You can define a new array by listing values separated with commas

```
// creates a 'donuts' array with three strings  
var donuts = ["glazed", "powdered", "jelly"];
```

But strings aren't the only type of data you can store in an array. You can also store numbers, Booleans... and really anything!

```
// creates a 'mixedData' array with mixed data types  
var mixedData = ["abcd", 1, true, undefined, null, "all the things"];
```

You can even store an array in an array to create a **nested array**!

```
// creates a `arraysInArrays` array with three arrays  
var arraysInArrays = [[1, 2, 3], ["Julia", "James"], [true, false, true, false]];
```

Nested arrays can be particularly hard to read, so it's common to write them on one line, using a newline after each comma:

```
var arraysInArrays = [  
  [1, 2, 3],  
  ["Julia", "James"],  
  [true, false, true, false]  
];
```

QUESTION

Select the valid arrays from the list below.

☐ ["pi" "pi" "pi" "pi"]

☐ [33, 91, 13, 9, 23]

☐ [null, "", undefined, []]

☐ [3.14, "pi", 3, 1, 4, "Yum, I like pie!"]

☐ true, 2, "Pie is good!"

☐ [33; 91; 13; 9; 23]

☐ {33, 91, 13, 9, 23}

Accessing Array Elements

Indexing

Remember that elements in an array are indexed starting at the position 0. To access an element in an array, use the name of the array immediately followed by square brackets containing the index of the value you want to access.

```
var donuts = ["glazed", "powdered", "sprinkled"];  
console.log(donuts[0]); // "glazed" is the first element in the `donuts` array
```

Prints: "glazed"

One thing to be aware of is if you try to access an element at an index that does not exist, a value of undefined will be returned back.

```
console.log(donuts[3]); // the fourth element in `donuts` array does not exist!
```

Prints: undefined

Finally, if you want to change the value of an element in array, you can do so by setting it equal to a new value.

```
donuts[1] = "glazed cruller";
```

QUESTION:

We've decided to replace some of donuts in the donuts array.

```
var donuts = ["glazed", "chocolate frosted", "boston cream", "powdered", "sprinkled",  
"maple", "coconut", "jelly"];  
  
donuts[2] = "cinnamon twist";  
  
donuts[4] = "salted caramel";  
  
donuts[5] = "shortcake eclair";
```

What does the donuts array look like after the following changes?

Exercise:

Create an array called `udaciFamily` and add "Julia", "James", and your name to the array.

Then, print the `udaciFamily` to the console using `console.log`.

```
var udaciFamily = ["Julia", "James", "Pargol"];  
for (var i = 0; i < udaciFamily.length; i++)  
{ console.log(udaciFamily[i]); }
```

Exercise:

Starting with this array of prices, change the prices of the 1st, 3rd, and 7th elements in the array.

```
var prices = [1.23, 48.11, 90.11, 8.50, 9.99, 1.00, 1.10, 67.00];
```

TIP: The 1st element of any array has an index of 0.

Afterwards, print out the prices array to the console.

Array Properties and Methods:

Array.length

You can find the **length** of an array by using its **length** property.

```
var donuts = ["glazed", "powdered", "sprinkled"];  
console.log(donuts.length);
```

Prints: 3

To access the length property, type the name of the array, followed by a period .(you'll also use the period to access other properties and methods), and the word length.

The length property will then return the **number of elements** in the array.

QUESTION

What is the length of the following inventory array?

```
var inventory = [  
  ["gold pieces", 25],  
  ["belt", 4],  
  ["ring", 1],  
  ["shoes", 2]  
];
```

So you can find length of an array, but what if you want to modify an array?

Thankfully, arrays have quite a few built-in methods for adding and removing elements from an array. The two most common methods for modifying an array are `push()` and `pop()`.

Push

You can use the `push()` method to add elements to the *end of an array*. example:
You can represent the spread of donuts using an array.

```
var donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar",  
"sprinkled"];
```

Then, you can *push* donuts onto the end of the array using the `push()` method.

```
donuts.push("powdered"); // pushes "powdered" onto the end of the `donuts` array
```

Returns: 7

Notice, with the `push()` method you need to pass the value of the element you want to add to the end of the array. Also, the `push()` method returns the length of the array after an element has been added.

```
var donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon  
sugar", "sprinkled"]; donuts.push("powdered"); // the `push()` method returns 7 because  
the `donuts` array now has 7 elements
```

Returns: 7

Pop:

Alternatively, you can use the `pop()` method to remove elements from the *end of an array*.

```
var donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar", "sprinkled", "powdered"];
```

```
donuts.pop(); // pops "powdered" off the end of the `donuts` array
```

```
donuts.pop(); // pops "sprinkled" off the end of the `donuts` array
```

```
donuts.pop(); // pops "cinnamon sugar" off the end of the `donuts` array
```

Returns: *"cinnamon sugar"*

donuts array: *["glazed", "chocolate frosted", "Boston creme", "glazed cruller"]*

With the `pop()` method you don't need to pass a value; instead, `pop()` will always remove the last element from the end of the array. Also, `pop()` returns the element that has been removed in case you need to use it.

```
var donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar", "sprinkled", "powdered"];
```

```
donuts.pop(); // the `pop()` method returns "powdered" because "powdered" was the last element on the end of `donuts` array
```

Returns: "powdered"

QUESTION:

We've decided to replace some of the donuts in the donuts array.

```
var donuts = ["glazed", "strawberry frosted", "powdered", "Boston creme"];
```

```
donuts.pop();
```

```
donuts.pop();
```

```
donuts.pop();
```

```
donuts.push("maple walnut");
```

```
donuts.pop();
```

```
donuts.push("sprinkled");
```

What does the donuts array look like after the following changes?

Splice

`splice()` is another handy method that allows you to add and remove elements from anywhere within an array.

While `push()` and `pop()` limit you to adding and removing elements from *the end of an array*, `splice()` lets you specify the index location to add new elements, as well as the number of elements you'd like to delete (if any).

```
var donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller"];
```

```
donuts.splice(1, 1, "chocolate cruller", "creme de leche"); // removes "chocolate frosted" at  
index 1 and adds "chocolate cruller" and "creme de leche" starting at index 1
```

Returns: ["chocolate frosted"]

donuts array after calling the `splice()` method: ["glazed", "chocolate cruller", "creme de leche", "Boston creme", "glazed cruller"]

Following is the syntax of splice() method:

```
arrayName.splice(arg1, arg2, item1, ....., itemX);
```

where,

- arg1 = Mandatory argument. Specifies the starting index position to add/remove items. You can use a negative value to specify the position from the end of the array e.g., -1 specifies the last element.
 - arg2 = Optional argument. Specifies the count of elements to be removed. If set to 0, no items will be removed.
 - item1,, itemX are the items to be added at index position arg1
- splice() method returns the item(s) that were removed.

`splice()` is an incredibly powerful method that allows you to manipulate your arrays in a variety of ways. Any combination of adding or removing elements from an array can all be done in one simple line of code.

QUESTION:

We've decided to replace some of the donuts in the donuts array.

```
var donuts = ["cookies", "cinnamon sugar", "creme de leche"];  
  
donuts.splice(-2, 0, "chocolate frosted", "glazed");
```

What does the donuts array look like after the following changes?

Exercise:

James was creating an array with the colors of the rainbow, and he forgot some colors.

The standard rainbow colors are usually listed in this order:

```
var rainbow = ["Red", "Orange", "Yellow", "Green", "Blue", "Purple"];
```

but James had this:

```
var rainbow = ["Red", "Orange", "Blackberry", "Blue"];
```

Using only the splice() method, insert the missing colors

into the array, and remove the color "Blackberry" by

following these steps:

- 1.Remove "Blackberry"

- 2.Add "Yellow" and "Green"

- 3.Add "Purple"

Exercise:

In the Harry Potter novels, children attending the Hogwarts School of Witchcraft and Wizardry belong to one of four houses: Gryffindor, Hufflepuff, Ravenclaw, or Slytherin.

Each year, the houses assemble a Quidditch team of seven players to compete for the coveted Quidditch Cup.

Consider the following array.

```
var team = ["Oliver Wood", "Angelina Johnson", "Katie Bell", "Alicia Spinnet", "George Weasley",  
"Fred Weasley", "Harry Potter"];
```

Create a function called `hasEnoughPlayers()` that takes the team array as an argument and returns true or false depending on if the array has at least seven players.

Exercise:

created a `crew` array to represent Mal's crew from the hit show Firefly.

```
var captain = "Mal";  
var second = "Zoe";  
var pilot = "Wash";  
var companion = "Inara";  
var mercenary = "Jayne";  
var mechanic = "Kaylee";  
  
var crew = [captain, second, pilot, companion, mercenary, mechanic];
```

Later in the show, Mal takes on three new crew members named "Simon", "River", and "Book".

Use the `push()` method to add the three new crew members to the `crew` array.

```
var doctor = "Simon";  
var sister = "River";  
var shepherd = "Book";
```

Array Loops

Once the data is in the array, you want to be able to efficiently access and manipulate each element in the array without writing repetitive code for each element.

For instance, if this was our original donuts array:

```
var donuts = ["jelly donut", "chocolate donut", "glazed donut"];
```

and we decided to make all the same donut types, but only sell them as *donut holes* instead, we could write the following code:

```
donuts[0] += " hole";  
donuts[1] += " hole";  
donuts[2] += " hole";
```

donuts array: ["jelly donut hole", "chocolate donut hole", "glazed donut hole"]

But remember, you have another powerful tool at your disposal, **loops**!

To loop through an array, you can use a variable to represent the index in the array, and then loop over that index to perform whatever manipulations your heart desires.

```
var donuts = ["jelly donut", "chocolate donut", "glazed donut"];

// the variable `i` is used to step through each element in the array
for (var i = 0; i < donuts.length; i++) {
    donuts[i] += " hole";
    donuts[i] = donuts[i].toUpperCase();
}
```

donuts array: ["JELLY DONUT HOLE", "CHOCOLATE DONUT HOLE", "GLAZED DONUT HOLE"]

In this example, the variable `i` is being used to represent the index of the array. As `i` is incremented, you are stepping over each element in the array starting from 0 until `donuts.length - 1`

The forEach() loop

Arrays have a set of special methods to help you iterate over and perform operations on collections of data. A couple big ones to know are the `forEach()` and `map()` methods.

The `forEach()` method gives you an alternative way to iterate over an array, and manipulate each element in the array with an inline function expression.

```
var donuts = ["jelly donut", "chocolate donut", "glazed donut"];

donuts.forEach(function(donut) {
  donut += " hole";
  donut = donut.toUpperCase();
  console.log(donut);
});
```

Prints:

*JELLY DONUT HOLE
CHOCOLATE DONUT HOLE
GLAZED DONUT HOLE*

Notice that the `forEach()` method iterates over the array without the need of an explicitly defined index. In the example above, `donut` corresponds to the element in the array itself. This is different from a `for` or `while` loop where an index is used to access each element in the array:

```
for (var i = 0; i < donuts.length; i++) {  
    donuts[i] += " hole";  
    donuts[i] = donuts[i].toUpperCase();  
    console.log(donuts[i]);  
}
```

Parameters

The function that you pass to the `forEach()` method can take up to three parameters.

The `forEach()` method will call this function once *for each* element in the array (hence the name `forEach`.) Each time, it will call the function with different arguments.

The `elementparameter` will get the *value* of the array element. The `index` parameter will get the *index* of the element (starting with zero). The `array` parameter will get a reference to the whole array, which is handy if you want to modify the elements.

Here's another example:

```
words = ["cat", "in", "hat"];
words.forEach(function(word, num, all) {
  console.log("Word " + num + " in " + all.toString() + " is " + word);
});
```

Prints:

Word 0 in cat,in,hat is cat
Word 1 in cat,in,hat is in
Word 2 in cat,in,hat is hat

Exercise:

Use the array's `forEach()` method to loop over the following array and add 100 to each of the values if the value is divisible by 3.

```
var test = [12, 929, 11, 3, 199, 1000, 7, 1, 24, 37, 4, 19, 300, 3775, 299, 36, 209, 148,  
169, 299, 6, 109, 20, 58, 139, 59, 3, 1, 139];
```

Map

Using `forEach()` will not be useful if you want to permanently modify the original array. `forEach()` always returns undefined. However, creating a new array from an existing array is simple with the powerful `map()` method.

With the `map()` [method](#), you can take an array, perform some operation on each element of the array, and return a new array.

```
var donuts = ["jelly donut", "chocolate donut", "glazed donut"];

var improvedDonuts = donuts.map(function(donut) {
  donut += " hole";
  donut = donut.toUpperCase();
  return donut;
});
```

donuts array: ["jelly donut", "chocolate donut", "glazed donut"]

improvedDonuts array: ["JELLY DONUT HOLE", "CHOCOLATE DONUT HOLE", "GLAZED
DONUT HOLE"]

Oh man, did you just see that?

The `map()` method accepts one argument, a function that will be used to manipulate each element in the array. In the above example, we used a function expression to pass that function into `map()`. This function is taking in one argument, `donut` which corresponds to each element in the `donuts` array. You no longer need to iterate over the indices anymore. `map()` does all that work for you.

Exercise:

Use the `map()` [method](#) to take the array of bill amounts shown below, and create a second array of numbers called `totals` that shows the bill amounts with a 15% tip added.

```
var bills = [50.23, 19.12, 34.01, 100.11, 12.15, 9.90, 29.11, 12.99, 10.00, 99.22, 102.20, 100.10, 6.77, 2.22];
```

Print out the new `totals` array using `console.log`.

TIP: Check out the `toFixed()` method for numbers to help with [rounding](#) the values to a maximum of 2 decimal places. Note, that the method returns a string to maintain the "fixed" format of the number. So, if you want to convert the string back to a number, you can **cast** it or convert it back to a number:

```
Number("1");
```

Returns: 1

Arrays in Arrays:

You could use an array of arrays that has the name of each donut associated with its position in the box.

```
Here var donutBox = [  
    ["glazed", "chocolate glazed", "cinnamon"],  
    ["powdered", "sprinkled", "glazed cruller"],  
    ["chocolate cruller", "Boston creme", "creme de leche"]  
];
```

If you wanted to loop over the donut box and display each donut (along with its position in the box!) you would start with writing a **for** loop to loop over each row of the box of donuts:

Next slide

```
var donutBox = [  
  ["glazed", "chocolate glazed", "cinnamon"],  
  ["powdered", "sprinkled", "glazed cruller"],  
  ["chocolate cruller", "Boston creme", "creme de leche"]  
];  
  
// here, donutBox.length refers to the number of rows of donuts  
for (var row = 0; row < donutBox.length; row++) {  
  console.log(donutBox[row]);  
}
```

Prints:

["glazed", "chocolate glazed", "cinnamon"]

["powdered", "sprinkled", "glazed cruller"]

["chocolate cruller", "Boston creme", "creme de leche"]

Since each row is an array of donuts, you next need to set up an inner-loop to loop over each cell in the arrays.

```
for (var row = 0; row < donutBox.length; row++) {  
  // here, donutBox[row].length refers to the length of the donut array currently being looped over  
  for (var column = 0; column < donutBox[row].length; column++) { console.log(donutBox[row][column]);  
    }  
}
```

Prints:

*"glazed"
"chocolate glazed"
"cinnamon"
"powdered"
"sprinkled"
"glazed cruller"
"chocolate cruller"
"Boston creme"
"creme de leche"*

Exercise:

Use a nested **for** loop to take the **numbers** array below and replace all of the values that are divisible by 2 (even numbers) with the string "even" and all other numbers with the

```
string var numbers = [  
    [243, 12, 23, 12, 45, 45, 78, 66, 223, 3],  
    [34, 2, 1, 553, 23, 4, 66, 23, 4, 55],  
    [67, 56, 45, 553, 44, 55, 5, 428, 452, 3],  
    [12, 31, 55, 445, 79, 44, 674, 224, 4, 21],  
    [4, 2, 3, 52, 13, 51, 44, 1, 67, 5],  
    [5, 65, 4, 5, 5, 6, 5, 43, 23, 4424],  
    [74, 532, 6, 7, 35, 17, 89, 43, 43, 66],  
    [53, 6, 89, 10, 23, 52, 111, 44, 109, 80],  
    [67, 6, 53, 537, 2, 168, 16, 2, 1, 8],  
    [76, 7, 9, 6, 3, 73, 77, 100, 56, 100]  
];
```

Objects in code

```
9  <script type="text/javascript">
10    var umbrella={
11      color : "pink",
12      isOpen : false,
13
14      open: function(){
15        if(umbrella.isOpen==true){
16          return "the umbrella is already opened!";
17        }else {
18          return "Pargol opens the umbrella";
19        }
20      }
21    };
22  </script>
```



NOTE:

`typeof` is an operator that returns the name of the data type that follows it:

```
typeof "hello" // returns "string"  
typeof true // returns "boolean"  
typeof [1, 2, 3] // returns "object" (Arrays are a type of object)  
typeof function hello() { } // returns "function"
```

Exercise:

Using the umbrella example from the previous slide, see if you can follow the example `open()` method and create the `close()` method. It's alright if you have trouble at first! We'll go into more detail later in this lesson.

TIP: Remember to put all of your object's properties and methods inside curly braces:

```
var myObject = { greeting: "hello", name: "Pargol" };
```

Also, remember that an object is just another data type. Just like how you would put a semicolon after a string variable declaration

```
var myString = "hello";
```

don't forget to put a semi-colon at the end of your object's declaration.

Object:

```
var sister = {  
  name: "Sarah",  
  age: 23,  
  parents: [ "alice", "andy" ],  
  siblings: ["julia"],  
  favoriteColor: "purple",  
  pets: true  
};
```

The syntax you see above is called **object-literal notation**. There are some important things you need to remember when you're structuring an object literal:

- The "key" (representing a **property** or **method** name) and its "value" are separated from each other by a **colon**
- The key: value *pairs* are separated from each other by **commas**
- The entire object is wrapped inside curly braces { }.

kind of like how you can look up a word in the dictionary to find its definition, the **key** in a **key:value** pair allows you to look up a piece of information about an object. Here's are a couple examples of how you can retrieve information about my sister's parents using the object you created.

```
// two equivalent ways to use the key to return its value  
sister["parents"] // returns [ "alice", "andy" ]  
sister.parents // also returns ["alice", "andy"]
```

Using `sister["parents"]` is called **bracket notation** (because of the brackets!) and using `sister.parents` is called **dot notation** (because of the dot!).

What about methods?

The sister object above contains a bunch of properties about my sister, but doesn't really say what my sister *does*. For instance, let's say my sister likes to paint. You might have a `paintPicture()` method that returns "Sarah paints a picture!" whenever you call it. The syntax for this is pretty much exactly the same as how you defined the properties of the object. The only difference is, the value in the key:value pair will be a function.

```
var sister = {  
  name: "Sarah",  
  age: 23,  
  parents: [ "alice", "andy" ],  
  siblings: ["julia"],  
  favoriteColor: "purple",  
  pets: true,  
  paintPicture: function() { return "Sarah paints!"; }  
};  
  
sister.paintPicture();
```

Returns: "Sarah paints!"

and you can access the name of my sister by accessing the `name` property:

`sister.name`

Returns: "Sarah"

Naming Conventions

1) Using “” for key

```
1 var person = {  
2   "name": "John",  
3   "age": 55,  
4   "1stChild": "James",  
5   "2ndChild": "Jarrod",  
6   "3rdChild": "Alexis"  
7 };
```

```
> var person = {  
    "name": "John",  
    "age": 55,  
    "1stChild": "James",  
    "2ndChild": "Jarrod",  
    "3rdChild": "Alexis"  
};
```

```
< undefined
```

```
> person["name"];
```

```
< "John"
```

```
> person["1stChild"];
```

```
< "James"
```

```
> person.name;
```

```
< "John"
```

```
> person.1stChild;
```

```
✖ Uncaught SyntaxError: Invalid or unexpected token VM499:1
```

Also don't use number for name of character : 1st

```
9    <script type="text/javascript">
10      var person = {
11        name : "John",
12        age : 55,
13        1stChild: "James",
14        2ndChild: "Jarrod",
15        3rdChild: "Alexis"
16      };
17    </script>
```

```
> var person = {
  name: "John",
  age: 55,
  1stChild: "James",
  2ndChild: "Jarrod",
  3rdChild: "Alexis"
};
```

✖ Uncaught SyntaxError: Unexpected identifier VM503:4

```
> |
```

2) Don't use space or using hyphens :

```
9  var garage = {
10    "fire truck": {
11      "color": "red",
12      "wheels": 6,
13      "operational": true
14    },
15    "race car": {
16      "color": "blue",
17      "wheels": 4,
18      "operational": false
19    }
20  };
```

```
> var garage = {
    "fire truck": {
      "color": "red",
      "wheels": 6,
      "operational": true
    },
    "race-car": {
      "color": "blue",
      "wheels": 4,
      "operational": false
    }
  };
<
```

```
undefined
```

```
> garage.fire truck;
```

✖ Uncaught SyntaxError: Unexpected identifier VM541:1

```
> garage.race-car;
```

✖ ▶ Uncaught ReferenceError: car is not defined(...) VM576:1

QUESTION:

Given the following user object, what would you use to print the value of the email property?

```
var user = {  
  email: "user@example.com",  
  firstName: "first",  
  lastName: "last"  
};
```

- ☐ `console.log(user[email]);`
- ☐ `console.log(email);`
- ☐ `console.log(user.email);`
- ☐ `console.log(user["email"]);`

Exercise:

Create a breakfast object to represent the following menu item:

The Lumberjack - \$9.95 eggs, sausage, toast, hashbrowns, pancakes

The object should contain properties for the name, price, and ingredients

Exercise: Using the given object:

```
var savingsAccount = {  
  balance: 1000,  
  interestRatePercent: 1,  
  deposit: function addMoney(amount) {  
    if (amount > 0) {  
      savingsAccount.balance += amount;  
    }  
  },  
  withdraw: function removeMoney(amount) {  
    var verifyBalance = savingsAccount.balance - amount;  
    if (amount > 0 && verifyBalance >= 0) {  
      savingsAccount.balance -= amount;  
    }  
  }  
};
```

add a `printAccountSummary()` method that returns the following account message:

Welcome! Your balance is currently \$1000 and your interest rate is 1%.

Exercise:

Directions:

Create an object called facebookProfile. The object should have 3 properties:

- 1.your name
- 2.the number of friends you have, and
- 3.an array of messages you've posted (as strings)

The object should also have 4 methods:

- 1.postMessage(message) - adds a new message string to the array
- 2.deleteMessage(index) - removes the message corresponding to the index provided
- 3.addFriend() - increases the friend count by 1
- 4.removeFriend() - decreases the friend count by 1

Exercise: Here is an array of donut objects.

```
var donuts = [  
  { type: "Jelly", cost: 1.22 },  
  { type: "Chocolate", cost: 2.45 },  
  { type: "Cider", cost: 1.59 },  
  { type: "Boston Cream", cost: 5.99 }  
];
```

Directions:

Use the `forEach()` method to loop over the array and print out the following donut summaries using `console.log`.

Jelly donuts cost \$1.22 each

Chocolate donuts cost \$2.45 each

Cider donuts cost \$1.59 each

Boston Cream donuts cost \$5.99 each

Browser JavaScript interface to HTML document

HTML document exposed as a collection of JavaScript objects and methods The

Document **O**bject **M**odel (DOM)

JavaScript can query or modify the HTML document

Accessible via the JavaScript global scope

Document Object Model (DOM)

A set of JavaScript objects that represent each element on the page

- each tag in a page corresponds to a JavaScript DOM object
- JS code can talk to these objects to examine elements' state

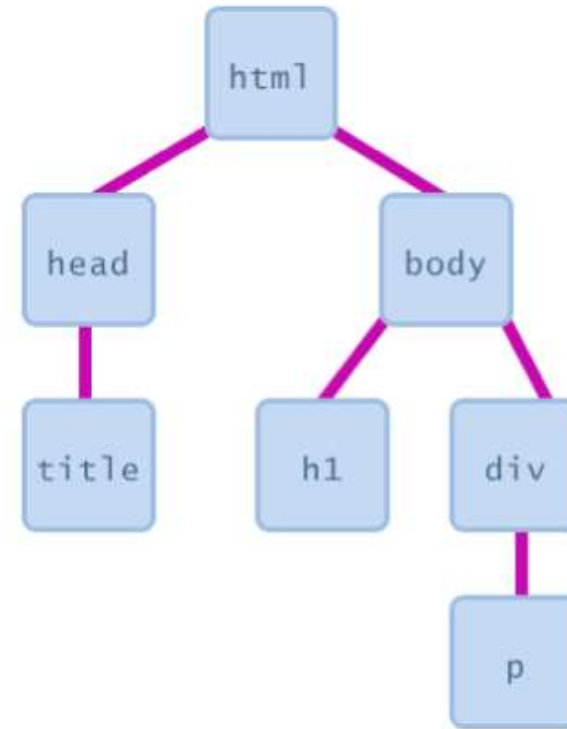
e.g. see whether a box is checked

- we can change state

e.g. insert some new text into a `div`

- we can change styles

e.g. make a paragraph red



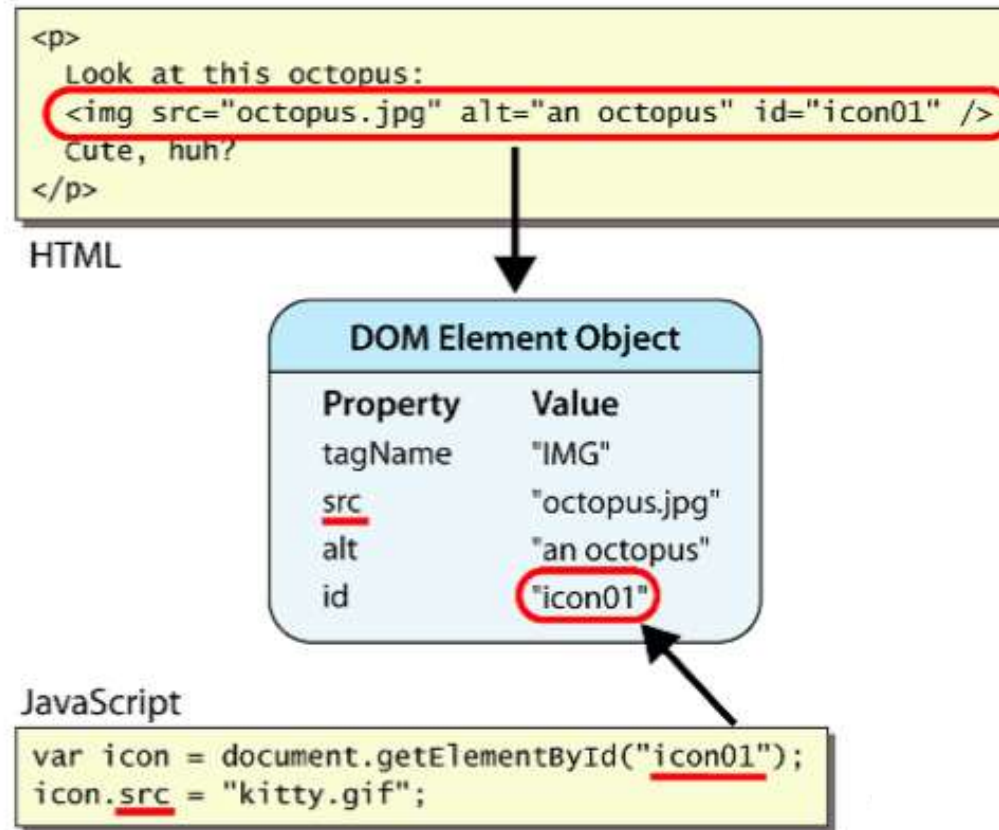
DOM element objects

access/modify the attributes of a DOM object with `objectName.attribute Name`


most DOM object attributes have the same names as the corresponding HTML attribute

`img` tag's `src` property

a tag's `href` property



Accessing an element: document.getElementById

<pre>var name = document.getElementById("id");</pre>	JS
<pre> <button onclick="changeImage();">Click me!</button></pre>	HTML
<pre>function changeImage() { var octopusImage = document.getElementById("icon01"); octopusImage.src = "images/kitty.gif"; }</pre>	JS
	output

`document.getElementById` returns the DOM object for an element with a given `id`

Finding HTML Elements

Method	Description
<code>document.getElementById(<i>id</i>)</code>	Find an element by element id
<code>document.getElementsByTagName(<i>name</i>)</code>	Find elements by tag name
<code>document.getElementsByClassName(<i>name</i>)</code>	Find elements by class name

Changing HTML Elements

Property	Description
<code>element.innerHTML = <i>new html content</i></code>	Change the inner HTML of an element
<code>element.attribute = <i>new value</i></code>	Change the attribute value of an HTML element
<code>element.style.property = <i>new style</i></code>	Change the style of an HTML element
Method	Description
<code>element.setAttribute(<i>attribute</i>, <i>value</i>)</code>	Change the attribute value of an HTML element

Adding and Deleting Elements

Method	Description
<code>document.createElement(<i>element</i>)</code>	Create an HTML element
<code>document.removeChild(<i>element</i>)</code>	Remove an HTML element
<code>document.appendChild(<i>element</i>)</code>	Add an HTML element
<code>document.replaceChild(<i>new</i>, <i>old</i>)</code>	Replace an HTML element
<code>document.write(<i>text</i>)</code>	Write into the HTML output stream

Adding Events Handlers

Method	Description
<code>document.getElementById(<i>id</i>).onclick = function(){<i>code</i>}</code>	Adding event handler code to an onclick event

Finding HTML Object:

Property	Description
document.anchors	Returns all <a> elements that have a name attribute
document.applets	Returns all <applet> elements (Deprecated in HTML5)
document.baseURI	Returns the absolute base URI of the document
document.body	Returns the <body> element
document.cookie	Returns the document's cookie
document.doctype	Returns the document's doctype
document.documentElement	Returns the <html> element
document.documentMode	Returns the mode used by the browser
document.documentURI	Returns the URI of the document
document.domain	Returns the domain name of the document server
document.domConfig	Obsolete. Returns the DOM configuration
document.embeds	Returns all <embed> elements
document.forms	Returns all <form> elements
document.head	Returns the <head> element
document.images	Returns all elements
document.implementation	Returns the DOM implementation

...

Finding HTML Element by Id

```
untitled
1 <!DOCTYPE html>
2 <html>
3   <body>
4     <h2>Finding HTML Elements by Id</h2>
5     <p id="intro">Hello World!</p>
6     <p>This example demonstrates the <b>getElementById</b> method.</p>
7     <p id="demo"></p>
8
9     <script>
10      var myElement = document.getElementById("intro");
11      document.getElementById("demo").innerHTML =
12        "The text from the intro paragraph is " + myElement.innerHTML;
13    </script>
14
15  </body>
16 </html>
17
18
```

Exercise :

Write an example to find all HTML elements with the same class name

Reacting to Events

A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element.

To execute code when a user clicks on an element, add JavaScript code to an HTML event attribute:

Examples of HTML events:

- When a user clicks the mouse
- When a web page has loaded
- When an image has been loaded
- When the mouse moves over an element
- When an input field is changed
- When an HTML form is submitted
- When a user strokes a key

Example:

```
1  <!DOCTYPE html>
2    <html>
3      <body>
4
5        <div onmouseover="mOver(this)" onmouseout="mOut(this)"
6          style="background-color:#D94A38;width:120px;height:20px;padding:40px;">
7          Mouse Over Me</div>
8
9      <script>
10        function mOver(obj) {
11          obj.innerHTML = "Thank You"
12        }
13
14        function mOut(obj) {
15          obj.innerHTML = "Mouse Over Me"
16        }
17      </script>
18    </body>
19  </html>
20
21
```

What is jQuery?

jQuery simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is a JavaScript toolkit designed to simplify various tasks by writing less code.

Here is the list of important core features supported by jQuery:

DOM manipulation:

The jQuery made it easy to select DOM elements, negotiate them and modifying their content by using cross-browser open source selector engine called **Sizzle**.

Event handling:

The jQuery offers an elegant way to capture a wide variety of events, such as a user clicking on a link, without the need to clutter the HTML code itself with event handlers.

AJAX Support:

The jQuery helps you a lot to develop a responsive and feature-rich site using AJAX technology

Animations:

The jQuery comes with plenty of built-in animation effects which you can use in your websites.

Lightweight:

The jQuery is very lightweight library - about 19KB in size (Minified and gzipped).

Cross Browser Support:

The jQuery has cross-browser support, and works well in IE 6.0+, FF 2.0+, Safari 3.0+, Chrome and Opera 9.0+

Latest Technology:

The jQuery supports CSS3 selectors and basic XPath

How to use jQuery?

There are two ways to use jQuery .



Local Installation

You can download jQuery library on your local machine and include it in your HTML code.



CDN Based Version

You can include jQuery library into your HTML code directly from Content Delivery Network (CDN).

jQuery Syntax

The jQuery syntax is tailor-made for **selecting** HTML elements and performing some **action** on the element(s).

Basic syntax is: **`$(selector).action()`**

- A \$ sign to define/access jQuery
- A (*selector*) to "query (or find)" HTML elements
- A jQuery *action()* to be performed on the element(s)

Examples:

`$(this).hide()` - hides the current element.

`$("p").hide()` - hides all <p> elements.

`$(".test").hide()` - hides all elements with class="test".

`$("#test").hide()` - hides the element with id="test".

What are Events?

All the different visitors' actions that a web page can respond to are called events.

An event represents the precise moment when something happens.

Examples:

- moving a mouse over an element
- selecting a radio button
- clicking on an element

The term "**fires/fired**" is often used with events. Example: "The keypress event is fired, the moment you press a key".

Here are some common DOM events:

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

jQuery Animations - The animate() Method

The jQuery `animate()` method is used to create custom animations.

Syntax:

```
$(selector).animate({params}, speed, callback);
```

The required `params` parameter defines the CSS properties to be animated.

The optional `speed` parameter specifies the duration of the effect. It can take the following values: "slow", "fast", or milliseconds.

The optional `callback` parameter is a function to be executed after the animation completes.

Finish