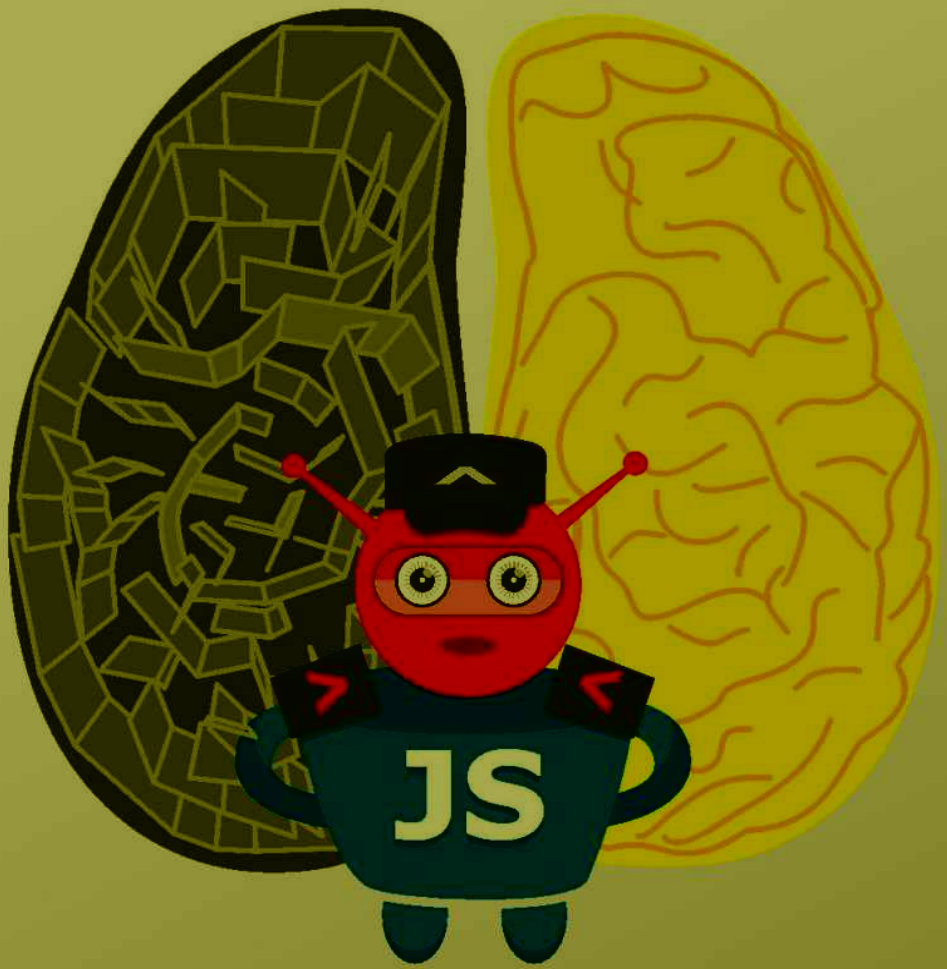
Learn JavaScript VISUALLY

Accelerated Learning Method
That Uses Science and Creativity
o Teach the Right Brain Non-Coders



Credits & Legal

AUTHOR
Ivelin Demirov

EDITOR
John Duncan

PAGE DESIGN
Jordan Milev

ILLUSTRATOR
Ivelin Demirov

PROOFREADER
Carol Dew

CREDITS
A book is a collaborative affair and it takes so many people to make it a reality and I would like to thank every Kickstarter backer who has helped to prepare this book for production.

I must acknowledge the help of the online JavaScript community, who have toiled in the background for many years to help make JavaScript the exciting programming language it has become

COPYRIGHT

NOTICE OF RIGHTS

TRADEMARKS

NOTICE OF LIABILITY

indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

PAPERBACK ISBN
978-1495233005

CONTACTS
jsvisually.com

Contents

Loops.

Arrays.

Arrays methods.

Functions.

Nesting.

The DOM node tree.

Manipulating the DOM.

Function statements
vs function expressions

Immediately-invoked
function expressions.

Scope.

Hoisting.

Closures.

Events.

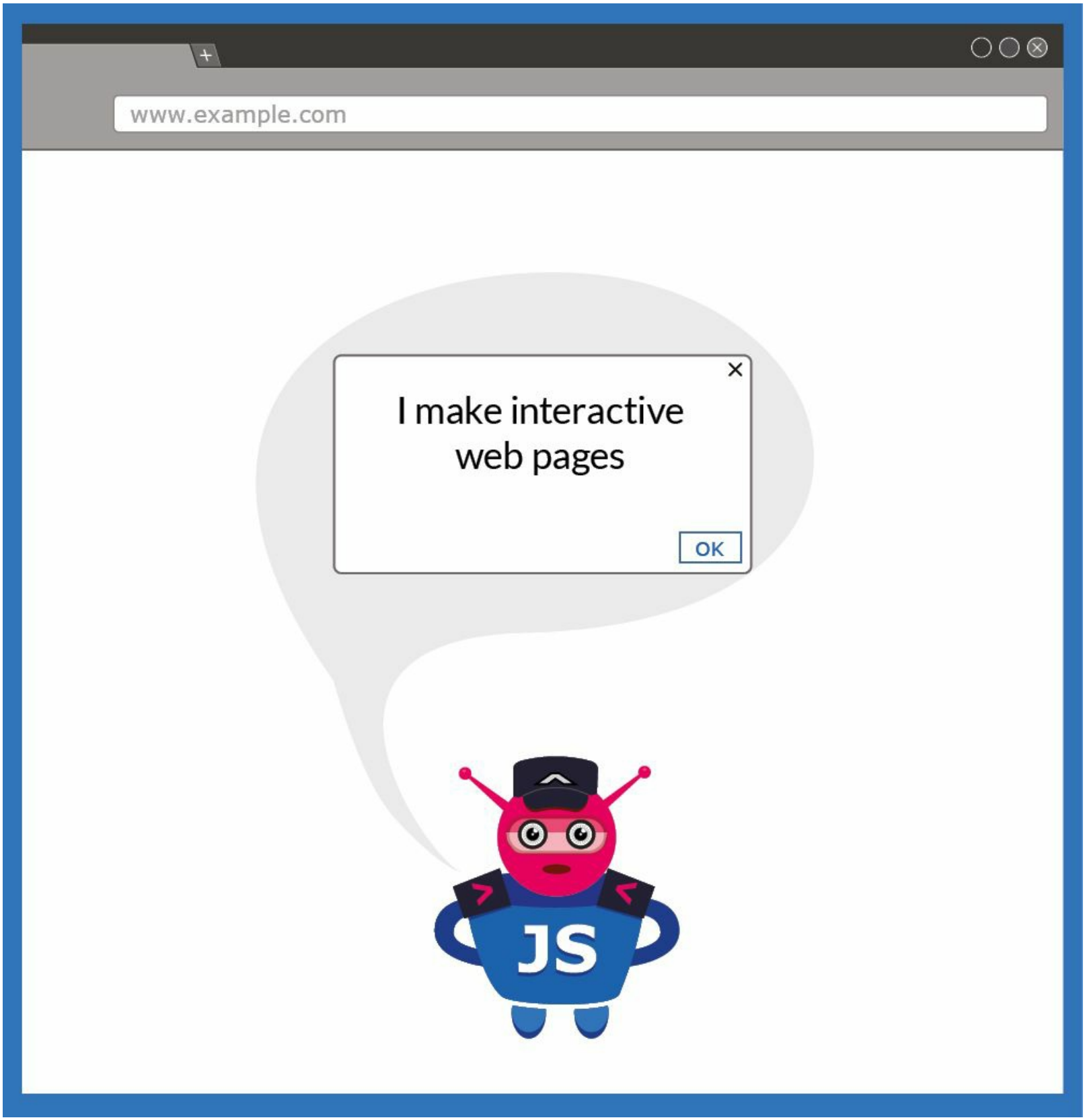
Objects.

Methods.

Constructors.

Prototypes.

Why JavaScript?



JavaScript is a dynamic computer programming language which allows us to make many dynamic actions on the client side via scripts. Because it is easy to learn and very functional, its popularity is high. JavaScript is used to validate forms, make games, and many more functions on the computer that is browsing the site. It is widely used as scripts, although it has an object oriented capabilities. Another good aspect is that it is also server

side which adds to its function as a companion to websites. JavaScript has dynamic typing and first class functions, which help its support not just object oriented, but also imperative and functional programming styles.JavaScript is also used outside the web in PDF documents, desktop widgets and more. This means that whether you are trying to create an interactive website, add functionality to your current one, or even have some scripts for your web server, the language will help you achieve your goals.

Since JavaScript is just a text, it does not need to be compiled – you can use any text editor of your choice to write or modify it. This contributes to its ease of use. There are also many available tools for you to debug, view, and even edit the JavaScript code from the web browser as you navigate a website.

If you are familiar with web development, you have probably heard of jQuery, Dojo and others. They are used to extend JavaScript and increase its potential and usefulness. They are libraries of code available for you to simplify some of the more complex commonly used code. After all, JavaScript is very powerful and fast, but writing everything from scratch can be very tedious.

Where do I put JavaScript?

Since JavaScript is used on websites, you would think you can just put it anywhere in the code of your site. Well, not quite.

You can add it in between the head tags *<head></head>*, and also on the body tags *<body></body>*.

It is very important to keep in mind that any JavaScript code must be enclosed by the script tag *<script></script>*.

Furthermore, you can also add scripts located anywhere else on the server, client, or even web, as long as you provide the clear path. These are known as external scripts. They are very handy when you have to reuse the same script on different sites.

All you have to do is place them somewhere where other sites can access and link to them.

The following code shows examples of where to put JavaScript code, and how to use both, internal and external code.

Just remember that they have to be in the scope of the script tags for them to work properly. The code will create a pop-up window every time you visit the site. It is very useful for providing important information to the user before anything else.

```
1. <!DOCTYPE html>
2. <html>
3.         <head>
4.         <title>My Title</title>
5. <script type="text/javascript"> alert(“It’s best if I sit low.”);</script>   <!--embed script-->
6. </head>
7. <body>
8.         <h1>Where do I put JavaScript?</h1>
9.         <!--lines 5 and 10 create a pop up with a message-->
10.        <script src = "main.js"></script><!--external script-->
11. </body>
12. </html>
```

```
1. alert("I pop-up from the external file!")
```

EXERCISE: 1
Where do I put JavaScript?

PROBLEM: Write a simple HTML file and link a ***javascript.js*** file with it.

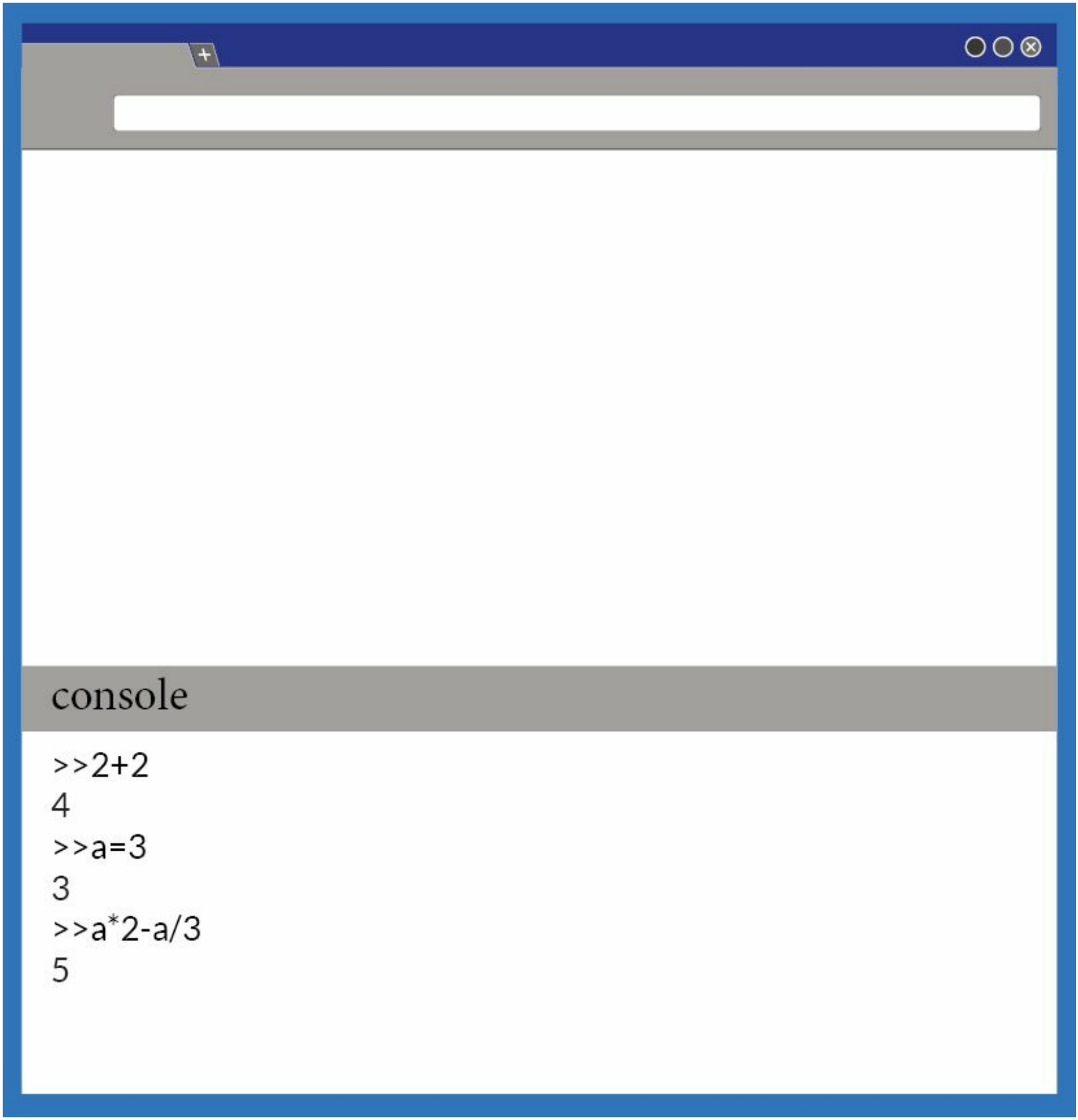
YOUR CODE: Write your code here and compare it with the answer below

ANSWER: jsvisually.com/1.html

How do I practice JavaScript?

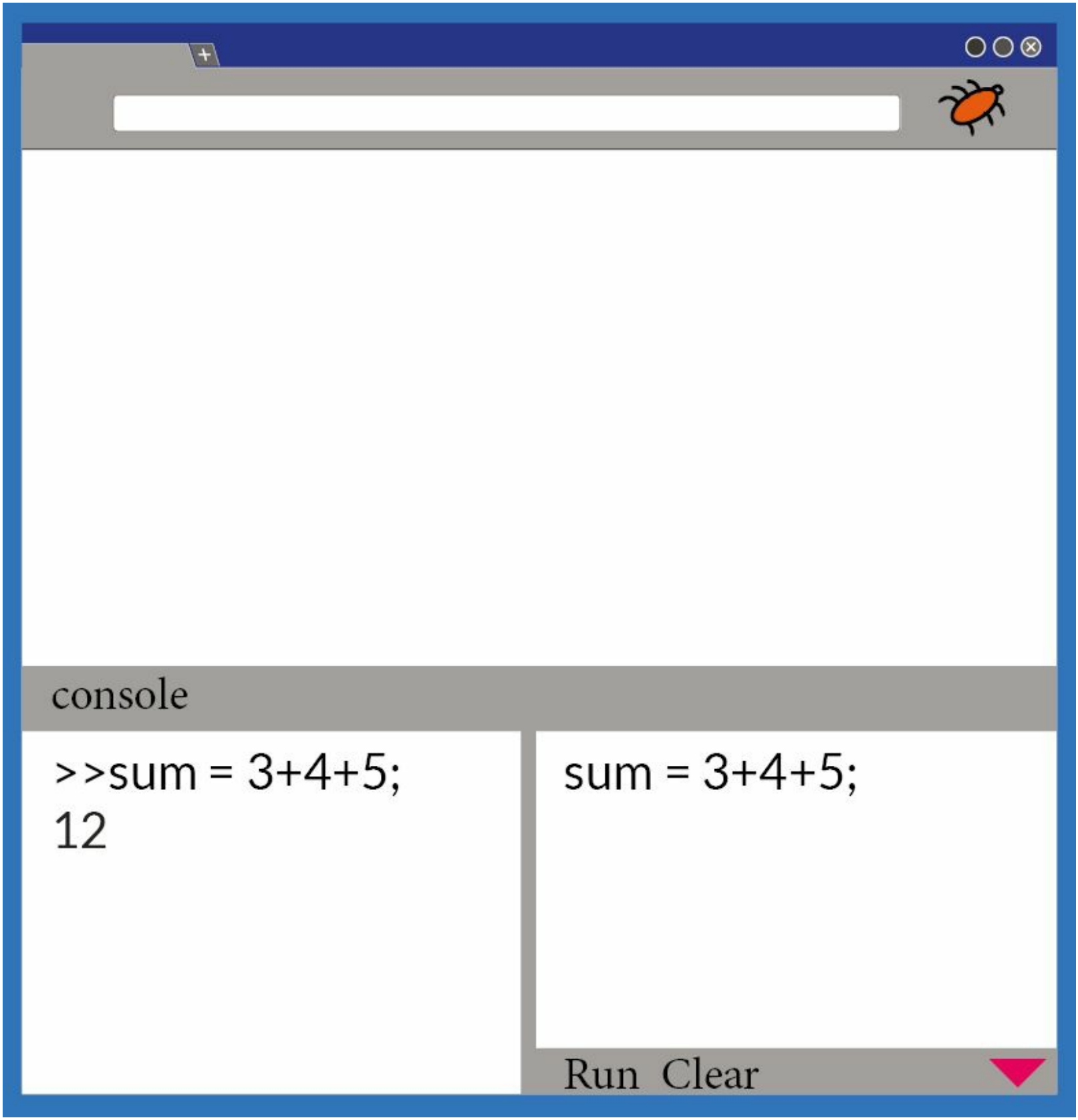
So far, JavaScript sounds interesting to you, right? Then you are probably wondering how to get started or what program you can use to start playing with JavaScript interactively. As we have mentioned before, there are many tools available, but JavaScript is so popular nowadays that most common web browsers have their own tool for JavaScript debugging, which you can use. Just look for the ***developer tools*** of your browser and open the JavaScript console and you will be able to test small bits of code right from your browser without having to install anything new. If you want more control, then you can install the Firebug extension.

In case you want to work locally with your code, then you can use specialized text editors such as Notepad++ or Sublime Text, which support many scripting and programming languages besides plain text. That means that you can use it to edit not just JavaScript, but the HTML of the site itself, if you have access to it. The both programs I mentioned are available free and are multiplatform.



in the console enter 2+2 and hit Enter

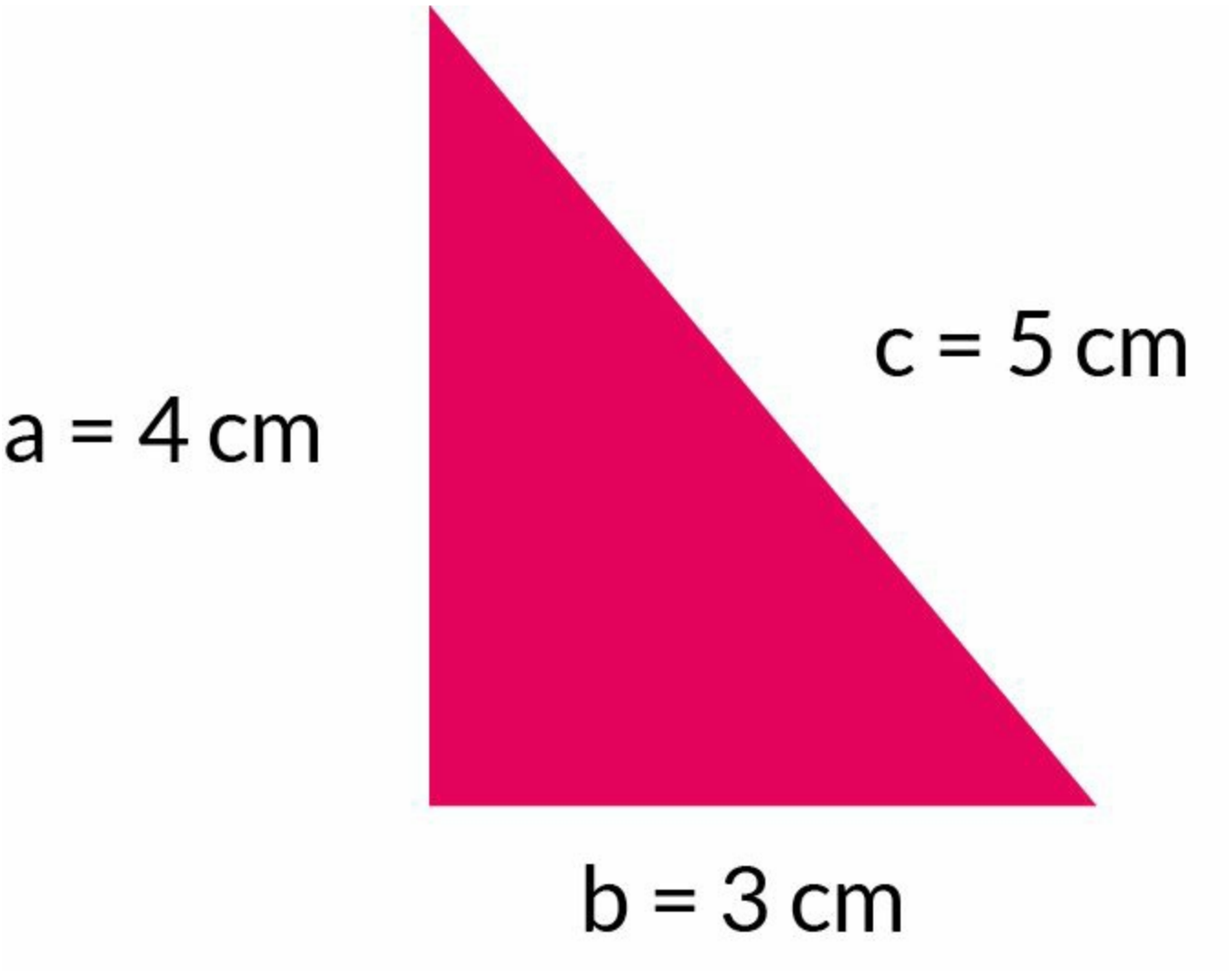
install the Firebug extension for even more control



click the red arrow to open/close the code panel

Exercise: 2
Practice JavaScript.

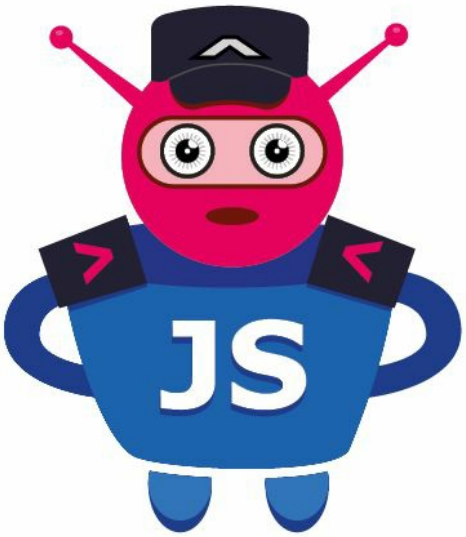
PROBLEM: Open the console of your browser and calculate the area of the right triangle below



ANSWER: jsvisually.com/2.html

The Interpreter /Compiler.

I am Private JS.
I will read, interpret/compile and execute your commands
from top to bottom and from left to right



I will run everything between the <script> tags

```
<script>
    alert("This line first");
    //ignore this line. It's a comment
    alert("This line second");
</script>
```

An interpreter or compiler is the software that interprets, compiles and executes code. JavaScript can be interpreted or compiled. Most modern browsers compile the code before execution.

However, this definition might not be enough for less experienced users in the world of programming. Both interpreters and compilers translate high-level language which is the programming languages we use that contains English words or from other languages, and turns this source code into machine language, which is what the computers actually understand. Machine code consists of 1's and 0's, also known as binary code.

There are a few differences between the interpreter and the compiler.
The interpreter translates one statement at the time, which reduces the time it takes to analyze the source code, yet the overall execution time is slower when compared to a compiler, which scans and translates the entire source code, thus taking longer time to analyze the code, but in return we get comparatively faster execution time.

Another difference is that the interpreter does not generate any intermediate object code, so it is more memory efficient, while the compiler does, and a need for more memory is an imperative.

Another key difference is in the debugging area – interpreters keep translating until the first error is met, while the compiler only displays it after it has finished with the entire source code, and so debugging becomes a little harder.

Some high-level programming languages use one over the other; for example, python uses the interpreter, while C uses compilers. However, JavaScript can currently use both, although at the beginning it was more of an interpreted language.

EXERCISE: 3
The Interpreter/Compiler.

PROBLEM: Modify the code to change the order of execution to the following:

first

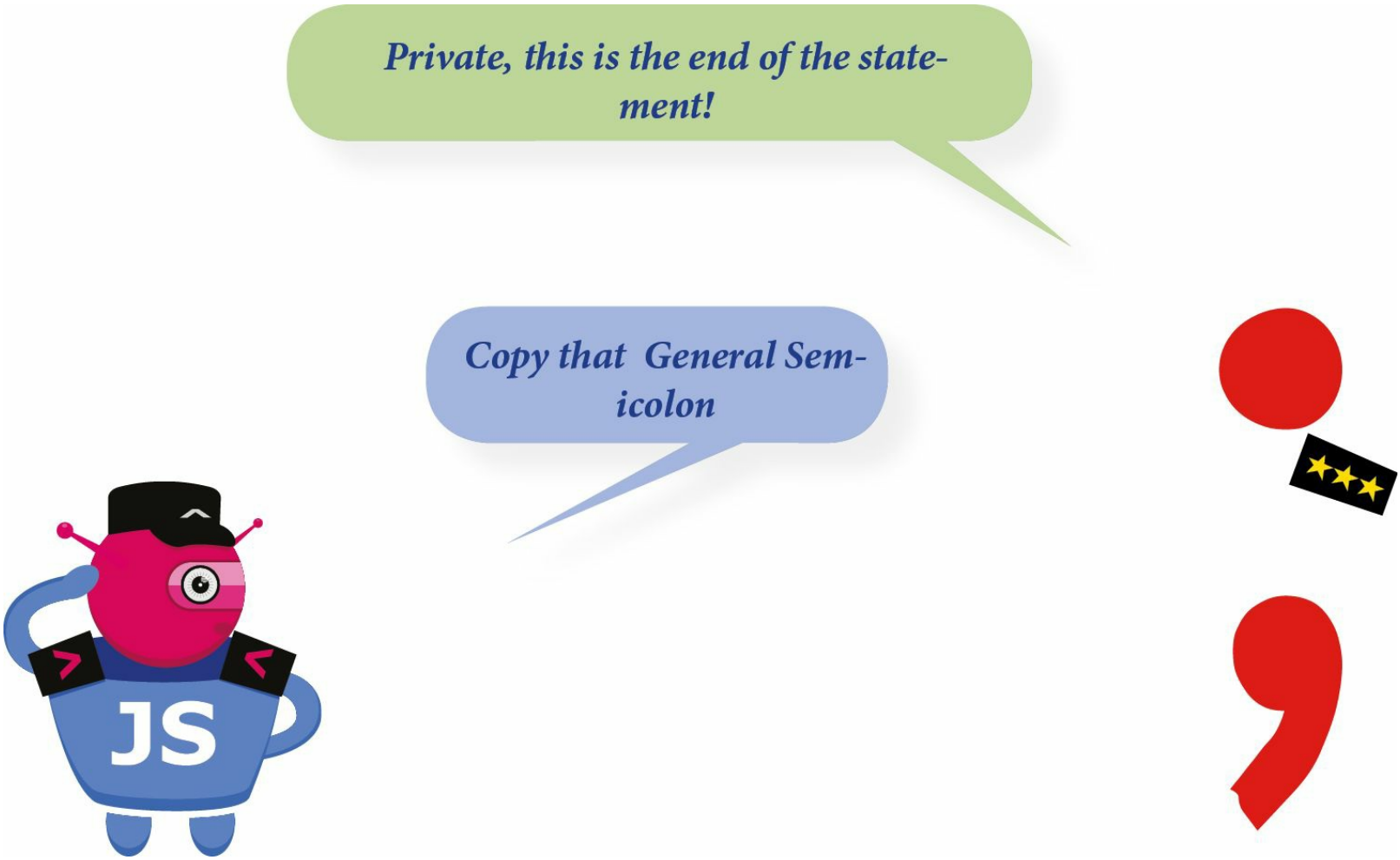
second

third

```
1. alert("execute this second");
2. alert("execute this first");
3. alert("execute this third");
```

ANSWER: jsvisually.com/3.html

Semicolons.



Semicolons are used to separate statements. As opposed to other programming languages, they are optional.

It is a good practice to consistently use them in your code.
Missing a semicolon might not give you an error, but your code might not run as expected.

When you have statements in one line, the first semicolon is required.

When you have statements separated by line breaks, semicolons are optional as the line break acts as one.

However, there are some cases in which you can break a statement into several lines and it will work like the example from line 8, while others will cause problems, as it is the case of the code on line 13.

You should also avoid putting a semicolon after a closing curly bracket with one exception -- when you are doing assignment statements. *var obj = {};*

```
1. // No semicolons after }:
2. if (...) {...} else {...}
3. for (...) {...}
4. while (...) {...}
5.
```

```
// BUT:
do {...} while (...);

// function statement:
function (arg) { /*do this*/ } // NO semicolon after }
```

You can put semicolons after the round bracket for if, for, while, or switch statement. Otherwise JavaScript will think it is an empty statement instead of it being part of a loop or condition.

```
if (0 === 1); { alert("hi") }

// equivalent to:

if (0 === 1) /*do nothing*/ ;
alert ("hi");
```

EXERCISE: 4
Semicolons.

PROBLEM: Add the missing semicolons.

```
1. x = 2y = 3
2. z = x + y
```

ANSWER: jsvisually.com/4.html

Brackets. Braces. Parentheses.

Brackets, braces and parentheses are grouping symbols which are always in pairs.
The brackets hold arrays, braces create object group statements, while parentheses supply parameters, group expressions, and execute functions.

While you can create arrays using either brackets or braces, they have their own characteristics, and by default brackets [] are the standard.

You can create arrays of numbers, strings, or even empty arrays.

```
1. //Array of numbers
2. var numArray = [1, 2, 3, 4, 5, 6, 7, 8, 9];
3. //Array of strings
4. var stringArray = ["John", "Paul"];
5. //Empty array
6. var emptyArray = [];
```

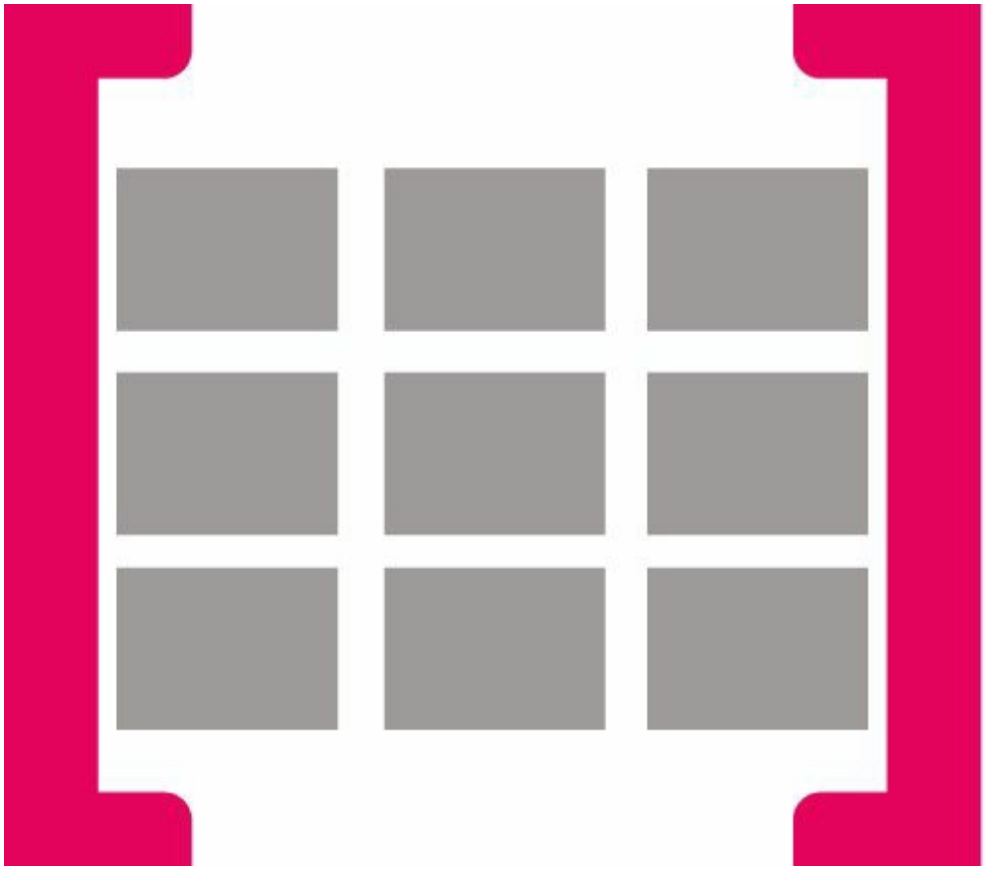
The most widely used for braces are grouping elements and creating objects:

```
1. //Create a new object
2. var myObj = {};
3. //Grouping statements
4. var a = function(){
5.     alert("Statement 1");
6.     alert("Statement 2");
7. };
```

Parentheses are used to supply parameter to functions, as you can see with the alert() functions in the example above. There are used to group the expressions and give proper order when doing complex math operations in one single line:

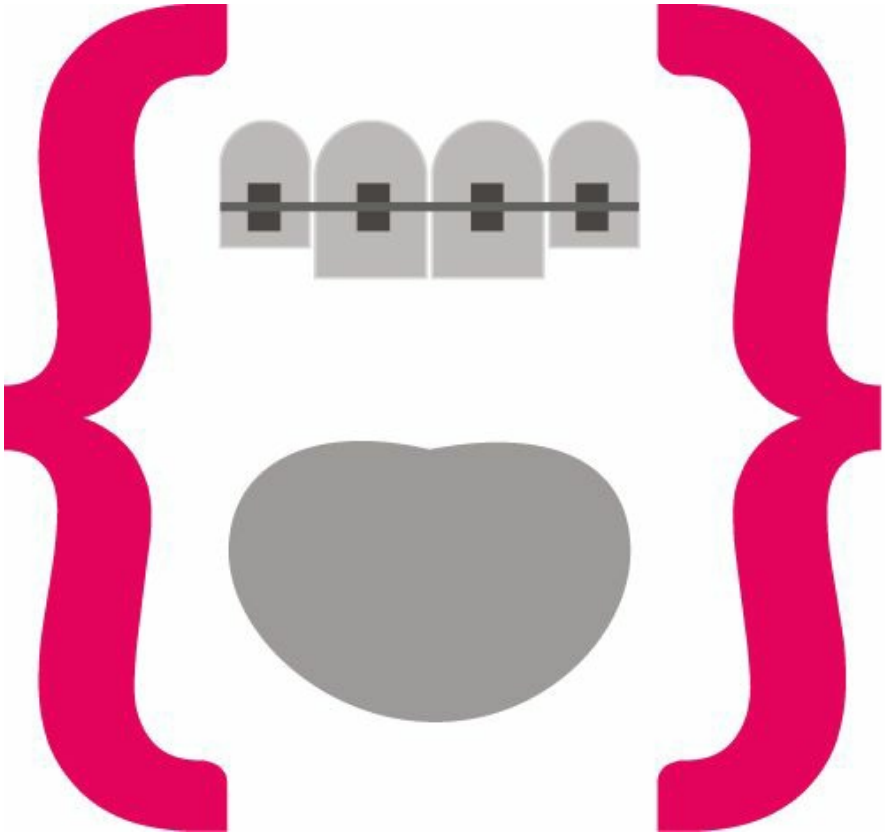
```
1. var a = (3 + 2) * 7; // a= 35
```

It is very common to see all three used in the same function in more complex code. The one thing to remember is that once you open any of them, they must be closed at some point and according to the scope of the program to avoid bugs.
These are one of the most common and hardest to spot when writing a large amount of codes for beginners, not just in JavaScript, but in many other programming languages.



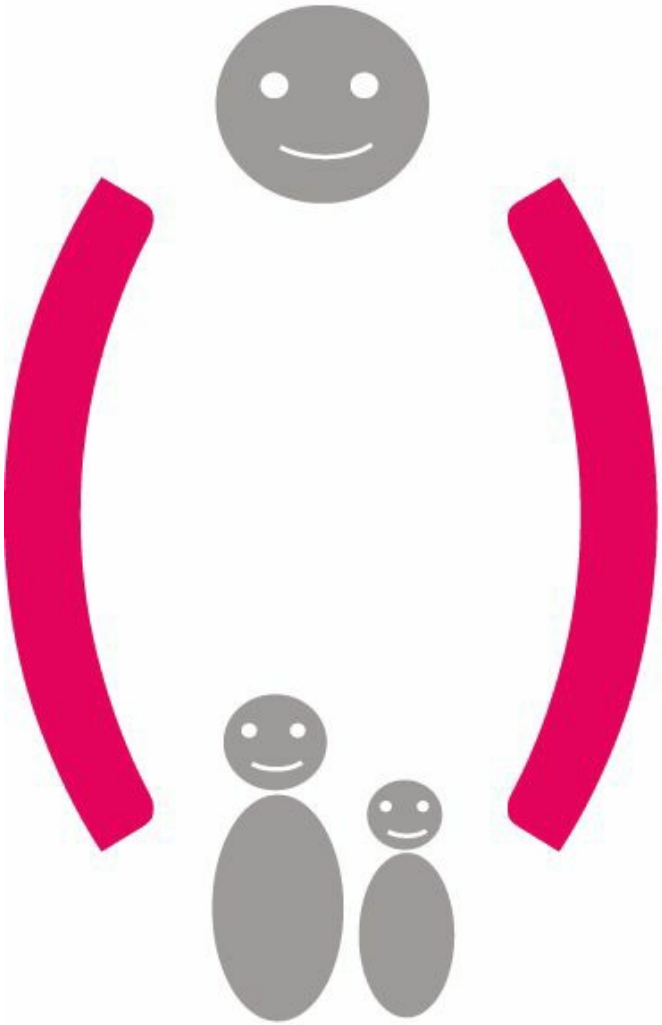
Brackets:
hold arrays

```
1. //array of numbers
2. var num = [1, 2, 3, 4, 5, 6, 7, 8];
3. //array of names
4. var names = ["John", "Paul"];
5. //empty array
6. var myArray = [];
```



Braces:
create objects
group statements

```
1. //create new object
2. var myObj = {};
3. //group statements
4. var a = function () {
5.  alert("statement 1");
6.  alert("statement 2");
7. };
```



Parentheses:

supply parameters
group expressions
execute functions

```
1. //supply parameters x and y
2. function myParents(x, y) {
3.  return x + y;
4. }
```

```
1. //group expressions to control the order
2. //of execution
3. var a = (3 + 2) * 7;          //a = 35
4. //execute functions
5. myParents(2, 3);          //5
```

EXERCISE 5
Brackets. Braces. Parentheses.

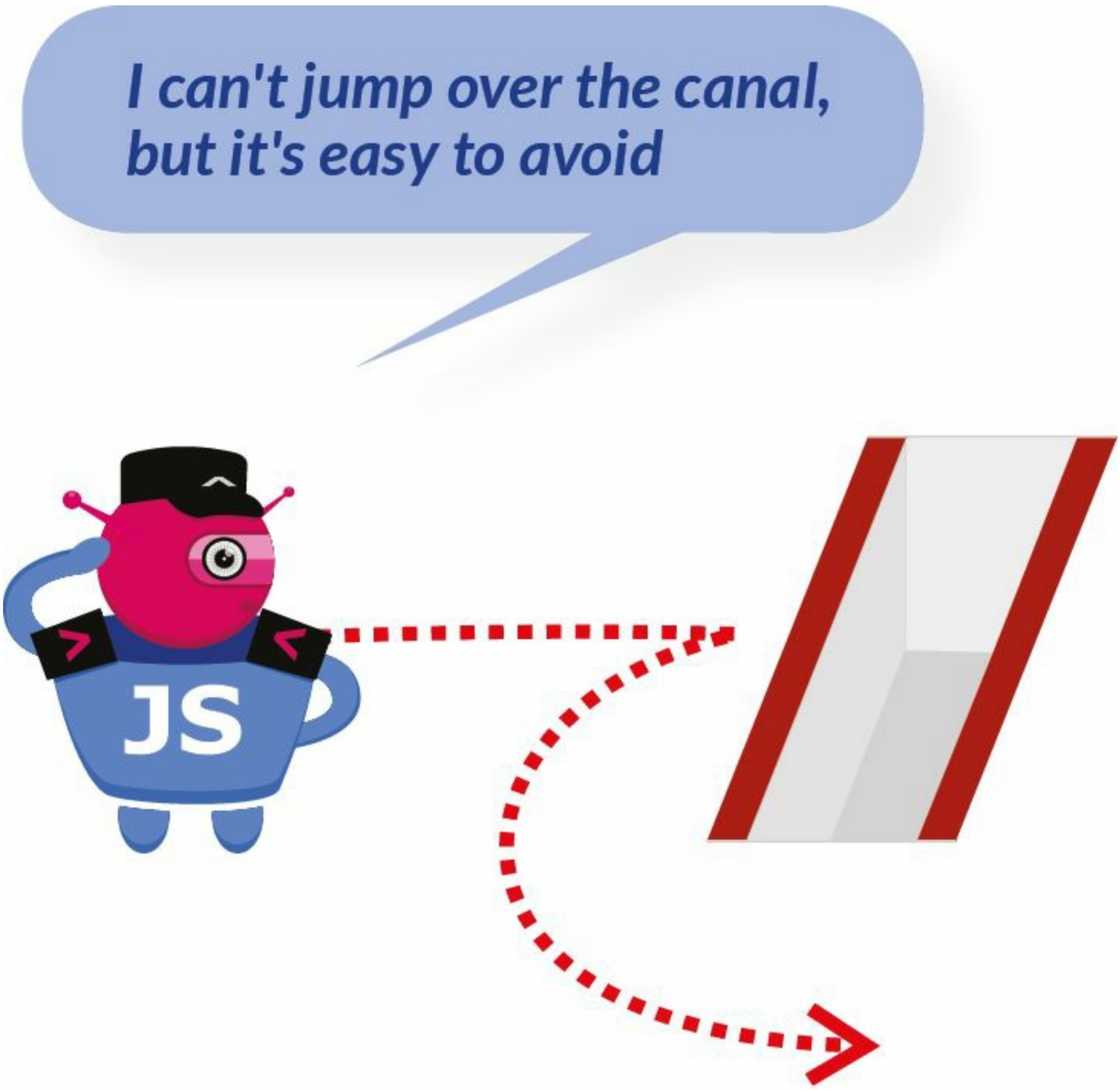
PROBLEM: Change the brackets to braces to resolve the code error.

Don't worry about the meaning yet, just identify the brackets and learn to type and test code.

```
1. var a = function () [
2.     alert("statement 1");
3.     alert("statement 2");
4. ];
5. SyntaxError
6. a();
```

ANSWER: jsvisually.com/5.html

Comments.



The interpreter will ignore a single line after the // symbols

```
<html>
  <head>
    <title>Single Comments</title>
  </head>
    <body>
      <script>
        //one comment per line

      </script>
    </body>
</html>
```

The interpreter will ignore anything in between /* and */


```
1. <html>
2.   <head>
3.     <title>Multiline Comments</title>
```

```
  </head>
  <body>
    <script>
      /* Long multiline comments
       are useful for "disabling" some
       part of your code
       var = 123;
       console.log("comments");
       */
       /* but are good for single line too */

    </script>
  </body>
</html>
```

The interpreter ignores comments, but they are great as they leave information for us and other coders. The interpreter will ignore a single line after the // symbols. The interpreter will also ignore anything in between /* and */.

Another use for JavaScript comments besides explaining the code and making it more readable is to also prevent code execution when you are testing alternative code.

You can place single line codes above the code or at the end of the line to leave your explanation.

```
// Change paragraph:
document.getElementById("myP").innerHTML = "My first paragraph.";

var x = 5; // Declare x, give it the value of 5
```

The use of single line comments is the most common, yet, for formal documentation purposes, block comments are the standard.

```
/*
The code below will change
the heading with id = "myHeader"
and the paragraph with id = "myParagraph"
in my web page:
*/
document.getElementById("myH").innerHTML = "My First Page";
document.getElementById("myP").innerHTML = "My first paragraph.";
```

When doing code testing, both single line comments and block comments can be used to prevent code execution by converting them into comments by using the appropriated method.

Blocking single line of code:

```
// document.getElementById("myH").innerHTML = "My First Page";
```

Blocking multiple lines of code:

```
/*
document.getElementById("myH").innerHTML = "My First Page";
document.getElementById("myP").innerHTML = "My first paragraph.";
*/
```

EXERCISE: 6
Comments.

PROBLEM: Comment out the following code in order to disable it from execution

```
1. var myObj = {};
2. var a = function () {
3.       alert("statement 1");
4.       alert("statement 2");
5. };
```

ANSWER: jsvisually.com/6.html

Data Types.

PRIMITIVE TYPES

Numbers

```
1  age = 26;                  //number
2  weight = 156.7;            //floating point
```

Strings

```
1  firstName = "Mark";        //double quote
2  famillyName = 'Twain';     //single quote
```

Booleans

```
1  z = true;
2  y = false;
```

SPECIAL DATA TYPES

null

```
1  age = null;    //was 26 before. null is an object
```

undefined

```
1  address;           //no value
2  a = function() {}; //no return statement
```

OBJECT TYPES

Object

```
1  employee = {name:"Joe", age:21, id:99};
2  function x() {};
```

Array

```
1  name = ["Joe", "Mike", "Freddy"];
```

Data types are the kind of values manipulated in a program. JavaScript variables can hold many different data types, such as numbers, strings, arrays, objects and many more that you will discover as you learn.

```
1. var age = 26;                                  // Number
2. var lastName = "Rodriguez";                    // String
3. var cars = ["Saab", "Volvo", "BMW"];           // Array
```

```
4. var x = {firstName:"John", lastName:"Doe"};   // Object
```

Not only in JavaScript but in any programming language, data types are very important as you will need to either declare them or be able to identify them when you are working with variables as you can't add a number to a string. Therefore, you work with variables using the same data types. You can convert one data type to another.

JavaScript evaluates expressions from left to right. So, while JavaScript uses the second expression for the data type to work with, if you have multiple sequences, then the results are different.

```
1. var x = 16 + "Blue";
```

JavaScript will read the above code as:

```
1. var x = "16" + "Blue";          //and thus you get x = 16Blue.
2. var x = 16 + 4 + "Blue";        //you get 20Blue.
3. var x = "Blue" + 16 + 4;        //you get Blue164.
```

JavaScript also has DYNAMIC TYPES. This means that the same variable can be used as more than one type, as you can see below.

```
1. var x;                  // x is undefined.
2. var x = 5;              // Now x is a Number.
3. var x = "John";     // Now x is a String.
```

Exercise: 7
Data types.

PROBLEM: Assign the following to a different letter from the alphabet:

your name,

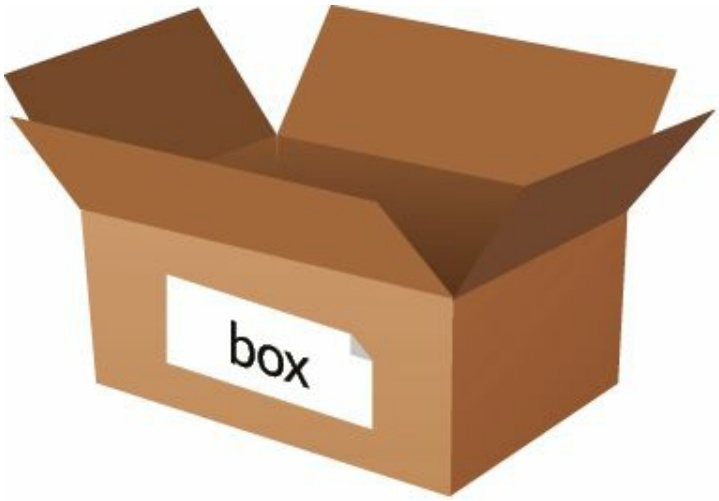
your age,

you like beer

make a comment with the data type for each one

ANSWER: jsvisually.com/7.html

Variables.



empty (undefined) variable declared by the name "box"

null is like a vacuum. It's an object, it's there, but it's nothing

```
1. //this variable is declared with a name but
2. //undefined as value
3. var box;
4. //null is an object
5. var box2 = null;
```

box now has the numeric value of 26

box2 and box3 refer to the same value in memory 24

```
//our variables have new values
//'box' is now initialized with the number 26,
//'box2' is not null anymore
var box = 26;
var box2 = 24;
//assign the same value to 2 different
//variables by separating them with comma
var box2 = 24,
        box3 = 24;
```

Each container can only have one object inside it, but can have many labels on it.

```
var name = true;
//the last assigned value replaces the previous
var name = "John";
//display the value of variable 'name'
console.log(name);
//John
```

Variable names (labels) are case sensitive and can only start with a letter (A-Z, a-z), an underscore (_) or $

```
1. //valid variable names:
2. var _myVar = "Yes";
3. var $myVar = "Yes";
4. var my$Var = "Yes";
5. var myVar$ = "Yes";
6. var myVar77 = "Yes";
7. var $myVAR_ = "Yes";
```

Variables are like labels that identify containers with information, and as the name implies, they are able to vary. Variables are used to store information and make it easy to reference them by giving them a name. The name given to the variable will remain the same, but you can always change the value or information that it contains. This means you could have a variable named “price” used to store the current price of an item; you could either assign a “null” value, which will create an empty variable, or assign any number value.

It is always in good practice to use descriptive names for your variables to make it easy to understand what they are used for in the program. This will come in handy for you or anyone else reading your code. It is also very important to use valid names for variables.

- Variables should not be enclosed in quotation marks.
- Variables can’t be a number or start with a number.
- Variables can’t be any of the JavaScript reserved keywords. However, they might contain keywords in the name.
- Variable names are case sensitive, so one should keep that in mind when

naming them.

- Variables can contain only letters, numbers, underscores, and the dollar sign.

When using numbers instead of strings, you can do math operations with them because for JavaScript they would be numbers.

```
1. var age = 26;
2. var currentYear = 2015;
```

Here you could have a third variable to calculate the birth year in the following way:

```
1. var birthYear = currentYear – age;
```

This is the same as saying bithYear = 2015 – 26. That's how JavaScript can handle an expression that only has variables in them to begin with.

In a similar fashion, you can have a variable calculating of changing its own value.

```
1. var originalNumber = 20;
2. var newNumber = originalNumber + 3;
```

Exercise: 8
Variables.

PROBLEM: John is a 25 year old male from United Kingdom.

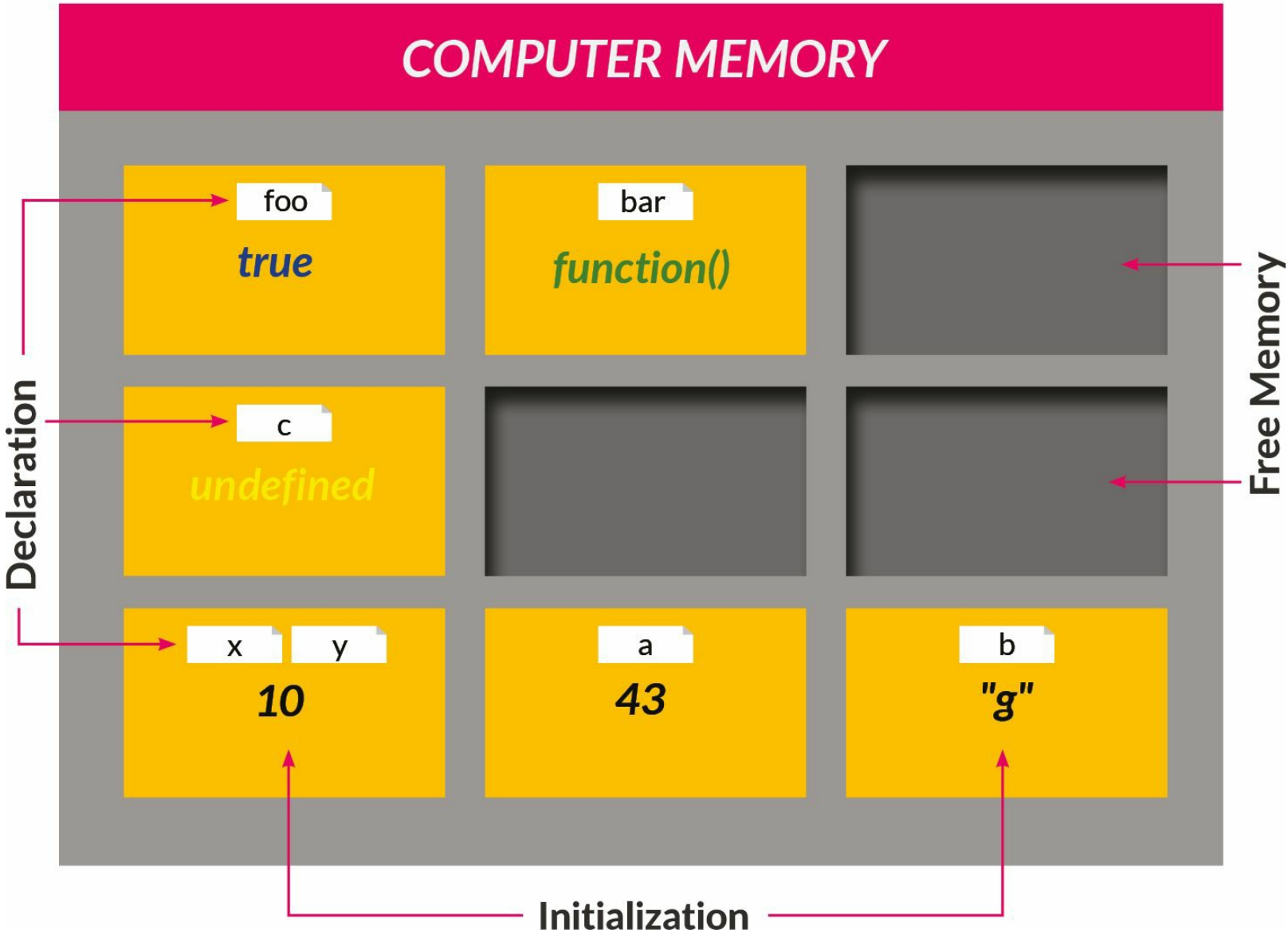
Assign all we know about him to a different variable so we can use it later.

ANSWER: jsvisually.com/8.html

Declaration and Initialization.

The simplest way to remember is to think of a computer memory as boxes, where declarations are the labels you assign to a box, and initialization is the value that you put on the box.

You can think of computer memory as an array of boxes



label

```
1   var foo = true;
2   var bar = function () { return 8; };
3
4   //'c' is declared but not initialized
5   var c;                          //undefined
6
7   //x and y refer (point) to the same value (10) in memory
8   var x = 10, y = 10;    //note the comma
9   var a = 43;
10  var b = "g";
```

value

Declaration is reserving a space in a memory. It can be for one or multiple variables and

objects. It is also very common to initialize them, at least the variables. Initializing means to assign an initial value that can later on either be changed or it can remain as a fixed value.

When you declare a variable and do not initialize it, the data type will be unidentified until you use the variable.

```
1. var x;       //It Is not initialized and therefore its data type is undefined.
```

Most of the time, you will see variables declared and initialized. Only in more complex and professional code where optimization is crucial and memory is very limited, you will see variables being declared but not initialized to reserve the space.

Exercise: 9
Declaration and initialization.

PROBLEM: Declare 5 different variables: a, b, c, d and e.

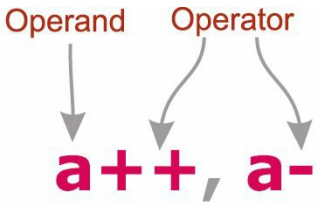
Assign a different type of value to each variable.

Leave one variable undefined.

ANSWER: jsvisually.com/9.html

Unary operators.

Operators that require ***single*** operand



a++, a--	take the value then add/subtract 1
++a, --a	add/subtract 1 then take the value
!a	logical opposite of **a**
+a, -a	positive/negative value of **a**

Unary operators require a single operand. They are easy to recognize as the format is the same as an operand and an operator.

Operand: It is the quantity on which an operation is to be done – basically, the variable.

Operator: It is the action that will be taken upon the operand.

The most common unary operators are:a++ , a--; which take the value and then add/subtract 1 to it and assigns it back to the variable. Similar to $a = a + 1;$ *or* $a = a - 1;$

- ++a, --a; which adds/subtract 1 then take the value.
- *!a; which takes the logical opposite of a.*
- +a,-a; which takes the positive/negative of a value.

The ++ is also known as the increment operator while the -- is known as the decrement operator.

One source of confusion with unary operators is the postfix (a++) and prefix(--a) increment or decrement operators. The following code will help you understand the difference which is crucial on loops, but not so noticeable outside of loops.

```
1. var a = 4;
2. var b = a++;          //First b will be 4, then it will be changed to 5. JavaScript works from left to right.
```

```
3. //now a has a value of 5.
4. var c = ++a;              //First a will be 6, then c will be assigned to 6
```

Exercise: 10
Unary operators.

PROBLEM: Using an unary operator, increase the value of a variable 'a'

OUTPUT: "The value of 'a' is: 11"

ANSWER: jsvisually.com/10.html

Binary operators.

+, -, *, /, % Arithmetic Operators

=, *=, /=, %=, +=,-=, <<=, >>=, >>>=, &=, ^=, |= Assignment Operators

&, |, ^, ~, <<, >>, >>> Bitwise Operators

==, !=, ===, !==, >, >=, <, <= Comparison Operators

&&, ||, ! Logical Operators

Binary operations are the most common and abundant of all JavaScript operators. You might not be familiar with all those symbols, so the following will be a brief explanation of some of the most used ones for your convenience.

Assume that A has a value of 10 and B has a value of 20.

1. Arithmetic Operators

The * Operator: Multiplies both operands. A * B will give 200

The / Operator: Divide numerator by dominator. B / A will give 2

The % Operator: Modulus Operator and remainder of after an integer division. B % A will give 0

The Addition operator (+): Adds two operands. A + B will give 30.

The Subtraction Operator (-): Subtracts second operand from the first. A - B will give -10

2. Assignment Operators

The = Operator: Simple assignment operator; Assigns values from right side operands to the left side operand. C = A + B will assign the value of A + B into C.

The *= Operator: Multiply AND assignment operator; it multiplies the right operand with the left operand and assign the result to the left operand. C *= A is equivalent to C = C * A.

The /= Operator: Divides AND assignment operator; It divides left operand with the right

operand and assigns the result to the left operand. C /= A is equivalent to C = C / A.

The %= Operator: Modulus AND assignment operator; It takes modulus by using two operands and assign the result to the left operand. C %= A is equivalent to C = C % A.

The += Operator: Adds AND assignment operator; It adds the right operand to the left operand and assign the result to the left operand. C += A is equivalent to C = C + A.

The -= Operator: Subtracts AND assignment operator; It subtracts right operand from the left operand and assign the result to the left operand. C -= A is equivalent to C = C – A.

3. Bitwise Shift Operators

The Left Shift Operator (<<): It moves all bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying by 2, shifting two positions is equivalent to multiplying by 4, etc. (A << 1) is 4.

The Signed Right Shift Operator (>>): It moves all bits in its first operand to the right by the number of places specified in the second operand. The bits filled in on the left depend on the sign bit of the original operand, in order to preserve the sign of the result. If the first operand is positive, the result has zeros placed in the high bits; if the first operand is negative, the result has ones placed in the high bits.

Shifting a value right one place is equivalent to dividing by 2 (discarding the remainder), shifting right two places is equivalent to integer division by 4, and so on. (A >> 1) is 1.

The Unsigned Right Shift Operator (>>>): This operator is just like the >> operator, except that the bits shifted in on the left are always zero. (A >>> 1) is 1.

4. Comparison Operators or Relational Operators

The Less-than Operator (<): Checks if the value of left an operand is greater than the value of the right operand; if yes, then condition becomes true. (A > B) is not true.

The Greater-than Operator (>): Checks if the value of the left operand is less than the value of the right operand; if yes, then condition becomes true. (A < B) is true.

The Less-than-or-equal Operator (<=): Checks if the value of the left operand is less than or equal to the value of the right operand; if yes, then condition becomes true. (A <= B) is true.

The Greater-than-or-equal Operator (>=): Checks if the value of the left operand is greater than or equal to the value of the right operand; if yes then condition becomes true. (A >= B) is not true.

5. Equality Operators

The Equals Operator (==): Checks if the value of two operands is equal or not; if yes, then condition becomes true. (A == B) is not true.

The Does-not-equals Operator (!=): Checks if the value of two operands is equal or not; if the values are not equal, then condition becomes true. (A != B) is true.

Exercise: 11
Binary operators.

PROBLEM: Add the value of two variables and display the result in the console using console.log() method.

OUTPUT: "8"

ANSWER: jsvisually.com/11.html

Bit operations.

AND

&	1	0
1	1	0
0	0	0

OR

|	1	0
1	1	1
0	1	0

XOR

^	1	0
1	0	1
0	1	0

NOT (unary)

~	1	0
	0	1

The Bit operators are also a part of the Binary Operators, but they are strictly used for logic operations as the ones you would find in discrete mathematics, manipulation of bits (zeros and ones) by bitwise operators.

1. Binary Bitwise Operators (&, ^, |):

- The ~ Operator: Called Bitwise NOT Operator. It is a unary operator and operates by reversing all bits in the operand. (~B) is -4.
- The ^ Operator: Called Bitwise XOR Operator. It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both. (A ^ B) is 1.
- The | Operator: Called Bitwise OR Operator. It performs a Boolean OR operation on each bit of its integer arguments. (A | B) is 3.
- The & Operator: Called Bitwise AND operator. It performs a Boolean AND operation on each bit of its integer arguments. (A & B) is 2.

2. Binary Logical Operators (&&, ||, !):

- The && Operator: Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. (A && B) is true.
- The || Operator: Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. (A || B) is true.
- The ! Operator: Called Logical NOT Operator. Use to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make false. !(A && B) is false.

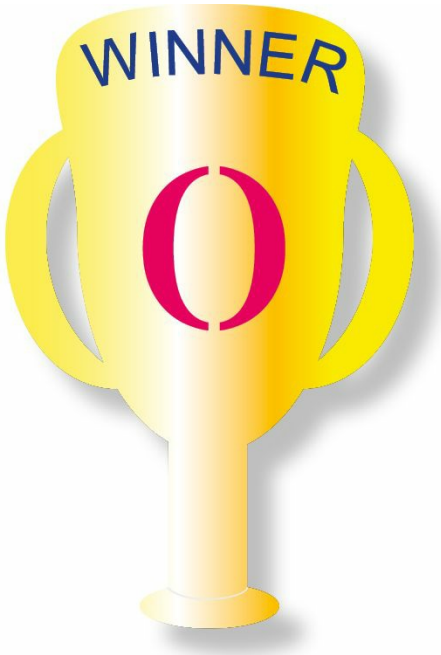
Exercise: 12
Bit operators.

PROBLEM: Add the numbers 2 and 3 using a bit operator and display the result in the console using the console.log() method

OUTPUT: "2"

ANSWER: jsvisually.com/12.html

Operator precedence.



*/%

+-

<< >> <= >=

2

3

4

HIGH PRIORITY

LOW PRIORITY

Multiplication	*Addition*	*Bit shifting*
Division	*Subtraction*	*Comparison*
Modulo		

Operator precedence determines the order in which operators are evaluated. Operators with grater precedence are evaluated first, just as in mathematics. By taking a look at the order of operations in arithmetic, we realize that exponents and roots go before multiplication and division, which have precedence over addition and subtraction.

```
1. 4 + 6 * 2    //returns 16.
```

Exercise: 13
Precedence.

PROBLEM: Modify the code so that the result is 10.5

```
1. var result = 33 -12 / 2;
2. document.write(result);
3. //27
```

ANSWER: jsvisually.com/13.html

Statements.

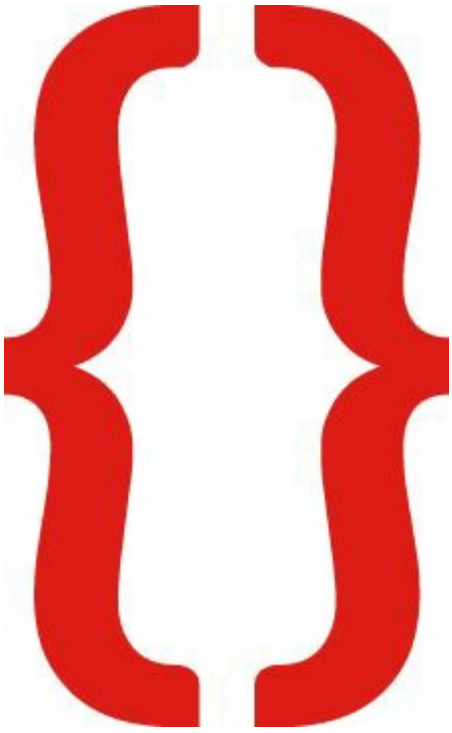
```
1.  //this is one long statement
2.  document.getElementById ("bold").innerHTML = "My text";
3.
4.  //control flow(conditional) statements
5.  if (true) {
6.          x = 8;          //{ x = 8; } is a block statement
7.  }
8.  else {
9.          x = 6;
10.  }
```

JavaScript is case sensitive

```
1. //JavaScript is case sensitive
2. //these variables are completely different
3. var myApple = "red";
4. var myapple = "red";
5. var MyApple = "red";
```

White space

```
1. //JavaScript ignores multiple spaces and tabs
2. //those are the same. The first space is required
3. varwhiteSpace ="empty";
4. var whiteSpace = "empty";
5. //white spaces are preserved in strings
6. var strings = "preservemyformat";
```



Code blocks

```
1. //functions are examples of code blocks
2. //or any group of statements delimited by braces
3. functionmyFunction () {
4.         var a = 3;
5.         alert ("my group" + a);
6. }
```

Statements are commands that are executed one by one in the same order they are written, unless they are conditional statements. We have been using them for a while and they are separated by semicolon. So, even declaring and assigning a value to a variable is a statement on its own.

JavaScript statements can be grouped together into blocks by using curly brackets. You will often see this in functions, and control flow statements.

```
1. while (x < 10) {
2.   x++;
3. }
```

JavaScript does not have block scope. Instead, the scope is the same as the function or script that precedes it. This is very important because they do not introduce a scope, so if you use the same variable name thinking that they have different scopes, then your script will have bugs. While standalone block statements are allowed, they do not do what you would normally expect, and thus, you might just write your statements without them.

```
1. varx = 1;
2. {
3.   varx = 2;
4. }
5. alert(x);      // outputs 2
```

Here the block statement is interfering with the code outside, which is why the output is 2 instead of 1.

The Conditional statements are the ones that run only when their condition is met. More details on that will follow in a later chapter.

Another type of statement is the Exception Handling. They are used to work around with exceptions by using the *throw* statement andt then handling them with the try … *catch* statement.

When you throw an exception, you specify the expression containing the value to be thrown. Also, you can throw any expression, string, number, Boolean, and even functions!

```
throw "Error2";   //String type
throw 42;         //Number type
throw true;       //Boolean type
throw {toString: function() { return "I'm an object!"; } };
```

The *try … catch* statement is a peculiar one. It marks a block of statements to try, and then you can specify one or more responses in case an exception is thrown. If that is the case, then the catch statement catches it. This is the best way to control your code for now exceptions.

```
try {
  throw "myException"     // generates an exception
}
catch (e) {
                                  // statements to handle any exceptions
  logMyErrors(e)              // pass exception object to error handler
}
```

The *finally* block is the last part, which is in charge of executing statements after the try and catch, but before the rest of the statements of your program or script. The finally block will be executed even if there is no catch block. Basically, the finally is used to make your program or script provide useful information as to why it failed, instead of just abruptly crashing.

```
openMyFile();
try {
   writeMyFile(theData);      //This may throw an error
}
catch(e) {
   handleError(e);                 // If we got an error we handle it
}
finally {
   closeMyFile();                 // always close the resource
}
```

Exercise: 14
Statements.

PROBLEM: Write a statement that has a case sensitive variable name containing a string with many white spaces

ANSWER: jsvisually.com/14.html

Expressions.

EXPRESSION TYPES

Arithmetic:	evaluates to a number
Logical:	evaluates to true or false
String:	evaluates to a character string
Object:	evaluates to an object

Expressions are sets of literals, variables and operators that resolve to a value. So if the line of code gives you a value, then it is an expression. There are two types of expressions: those that assign a value to a variable and those that simply have a value.

```
1. var z = 8;       //First type of expression.
2. 6 + 2;           // Second type of expression.
```

JavaScript has the four expression categories:

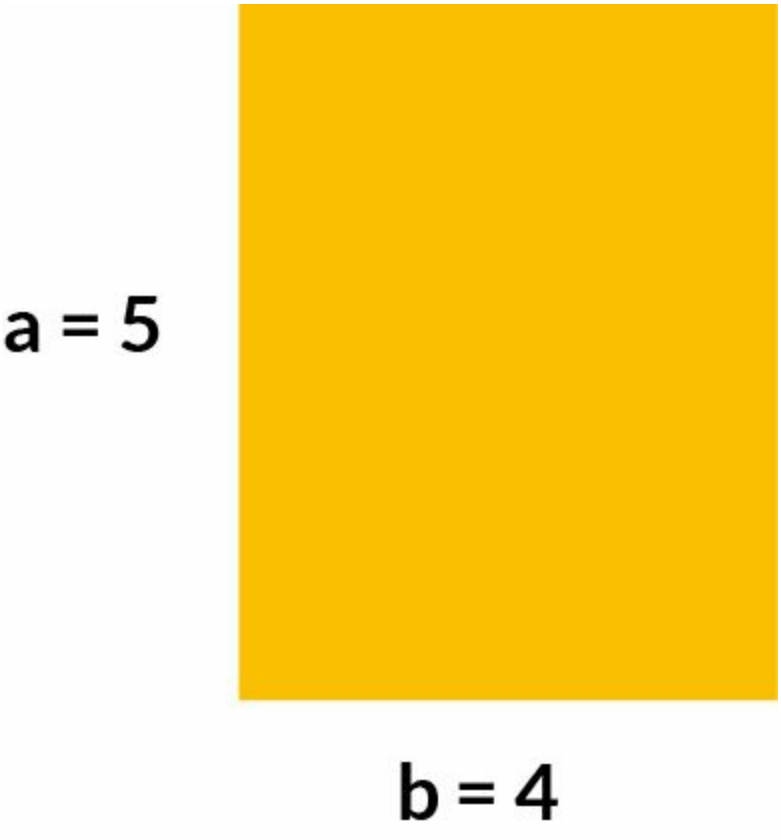
1. Arithmetic: They evaluate to a number and generally use arithmetic operators.

2. String: They evaluate to a character string and generally use string operators.
3. Logical: They evaluate to an object and normally involve logical operators.
4. Object: They evaluate to an object.

```
1. //Arithmetic expression
2. var a = 3;
3. //Logical expression
4. var b = true;
5. //String expression
6. var c = “String”;
7. //Object expression
8. var apple = newApple(“Machintosh”);
```

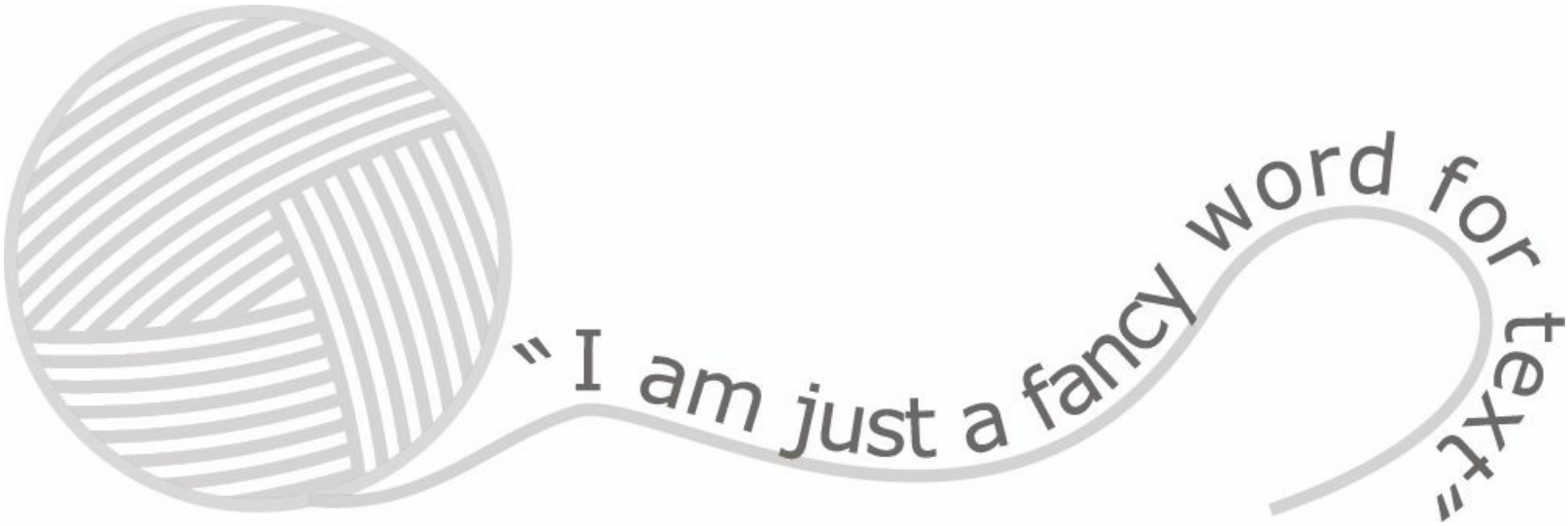
Exercise: 15
Expressions.

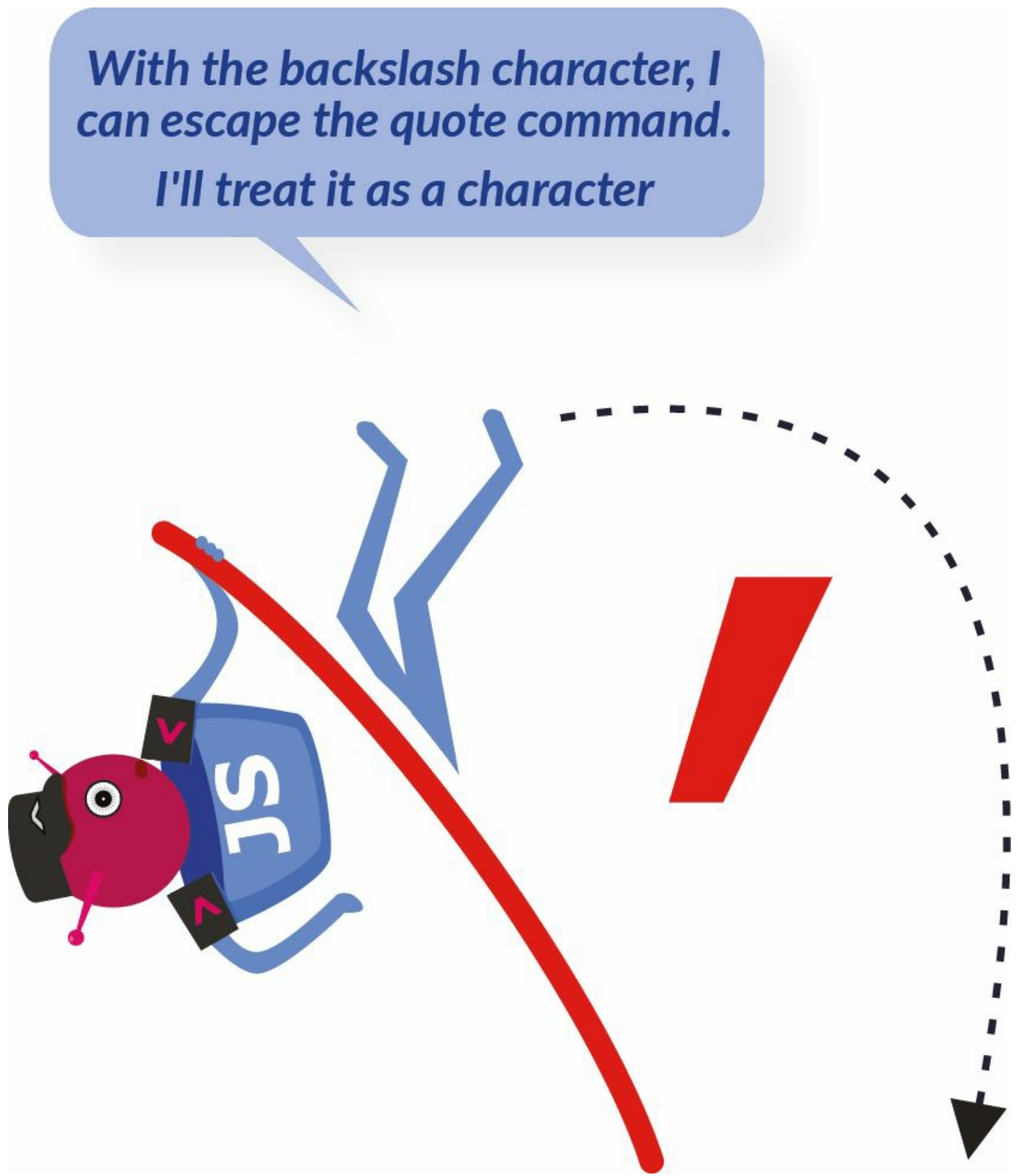
PROBLEM: Write an expression that represents the perimeter of the rectangle below. The perimeter of any rectangle is the sum of the lengths of its sides.



ANSWER: jsvisually.com/15.html

Strings.





Strings are groups of characters that form regular text. To make a string, you can write any set of characters in quotes, whether they are single or double quotes.

```
var fullName = "John Doe"
var fullName2 = 'Mary Jane'
```

It is also possible to use both -- this would be to add quotations inside a string. The only rule for this is that if you use one type of quotation for the string, then you must use the second kind for the quotation inside the string.

```
var question = "What are 'quotations'?"
```

Since strings are so widely used, there are many properties available for them. If you need to find the length of a string, then you use the .length property, which will return an integer representing the number of characters in the string.

In case that you must use the same quotation mark, you can use the \ escape character. The backslash escape character can be used to turn special characters into a string, and also to insert characters in a string.

```
varx = 'It\'s alright';
vary = "We are the so-called \"Vikings\" from the north."
```

As for the special characters we have:

Code	Outputs
\'	single quote
\"	double quote
\\	backslash
\n	new line
\r	carriage return
\t	tab
\b	backspace
\f	form feed

Strings also have a wide range of methods to work with -- you can find the index of a specific character by using charAt(); you can also split strings into arrays of substrings with split(), or slice it with slice() to obtain a part of a string and turn it into a new string.

You might have noticed by now that strings share some similarities to arrays.
In a way you could think of them as arrays of characters strictly.

Strings like arrays are indexed and start at 0.
Also, white spaces do count as characters. Most string methods work directly with indexes, whether it is searching, replacing, or any other modifications.

Exercise: 16
Strings.

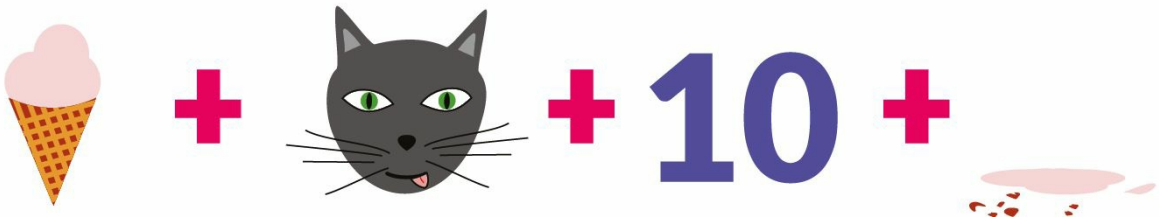
PROBLEM: Use the console.log() method to display the following message exactly as it appears

```
1. Hello my friend,
   this message and its meaning
            was formatted using "JavaScript'©
```

ANSWER: jsvisually.com/16.html

Concatenation.

cone + cat + ten + ate



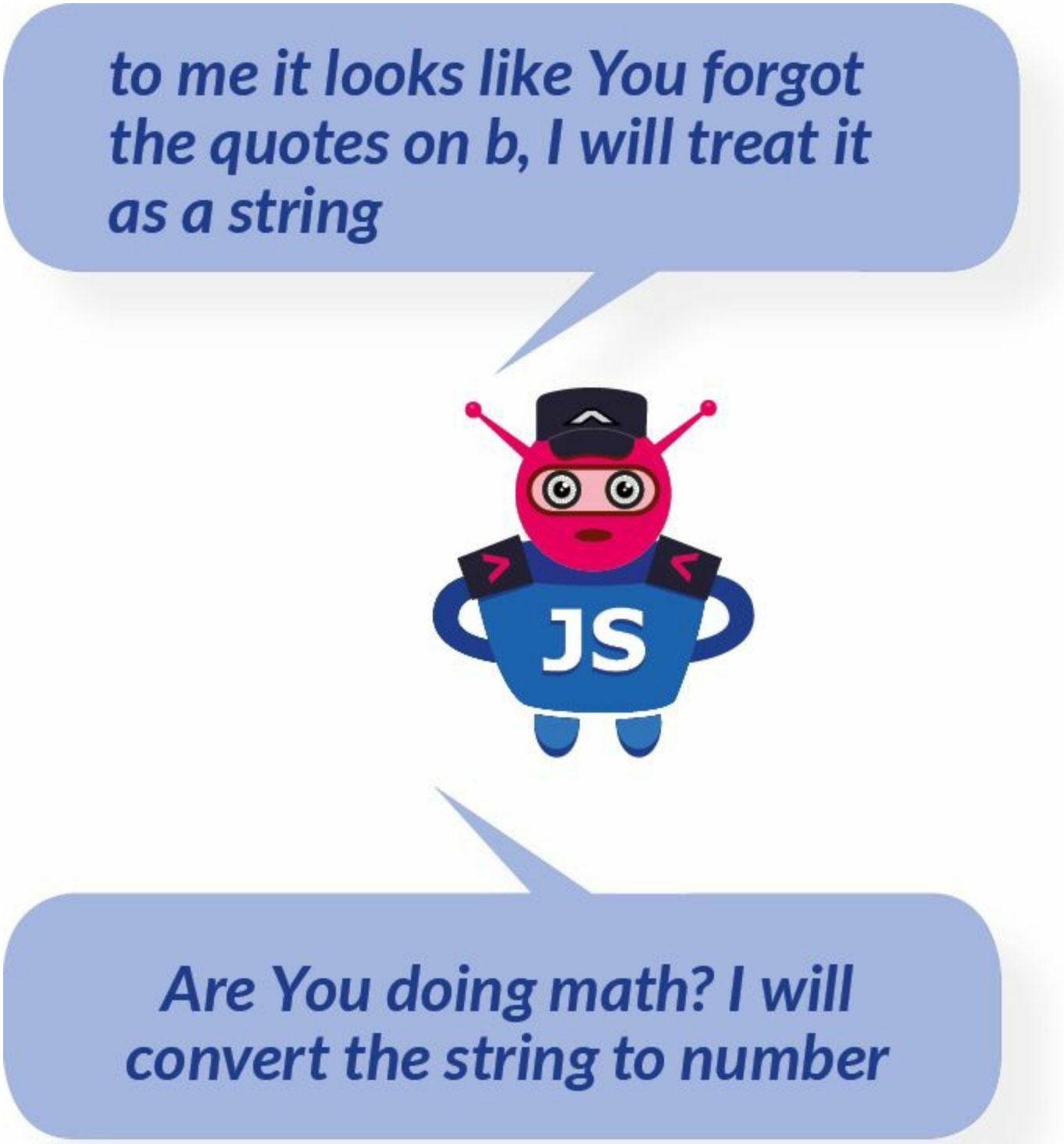

```
1. //concatenation of strings
2. var conString = "con" + "cat" + "enate";
3. //the result is: concatenate
```

```
1. //numbers in quotes become strings
2. var a = "1";
3. var b = "2";
4. var c = a + b;
5. //the result is a string: c = "12"
```

```
1. //concatenation is different from addition
2. var a = 1;
3. var b = 2;
4. var c = a + b;
5. //the result is a number: c = 3
```

```
1. //one string, makes everything string
2. var a = "1";              //this is a string
3. var b = 2;                //this is a number
4. var c = a + b;
5. //The result is a string: c = 12
```

```
1. //concatenation is only when we have "+" sign
2. var a = "3";              //this is a string
3. var b = 2;                //this is a number
4. var c = a * b;
5. //The result is a number: c = 6
```



Concatenation is the word for joining strings together. Joining strings together is rather simple, since we already use the operator needed for it with math. When you want to join two or more strings, all you have to do is use the plus sign. Keep in mind that if you need space on the new string, you will have to add it manually by concatenating a white space.

```
1. var conString = “con”+”cat”+”enate”;
```

There are also two other ways to concatenate in JavaScript:

- By using the assignment operator (+=)
- By using the built-in concat method

```
var str1 = "Welcome";
str1 += " To";
str1 += ” Javascript”;

var str1 = "".concat("Welcome ","To ","Javascript");
```

For the concat() method, the existing string is not changed. Instead, a new one is created with the content of the texts joined together. However, when using + and +=, you are changing the existing string, unless you assign the concatenation to a new variable manually.

You can turn numbers to string to concatenate them. However, if you have a number and a string, JavaScript will treat both of them as strings.

```
var a = “1”;
var b = 2;
var c = a + b;
//The result will be “12” as the value of the variable c.
```

In case that you have another operand other than +, the result will be different.

```
var c = a * b;
//From the previous example, the result will be 2 instead of “12”.
```

Exercise: 17
Concatenation.

PROBLEM: Join "java" and "script" with the number 13.

make sure you have a space before the number.

Print it in the console.

ANSWER: jsvisually.com/17.html

Reserved words.

JavaScript has some identifiers that are reserved words and they cannot be used as variables or function names, as they will be always interpreted as reserved words. Because some of the reserved words could be easily part of a name, if you insist on using one of them in your naming of variables or functions, then you should add more to the name, following the naming convention explained when introducing the variables.

Here are the reserved words:

abstract	arguments	boolean	break	byte
case	catch	char	class	const
continue	debugger	default	delete	do
double	else	enum	eval	export
extends	false	final	finally	float
for	function	goto	if	implements
import	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return
short	static	super	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

Furthermore, there are JavaScript objects, properties, and methods with reserved names that you should avoid using, as they are built in into JavaScript:

Array	Date	eval	function	hasOwnProperty
Infinity	isFinite	isNaN	isPrototypeOf	length
Math	NaN	name	Number	Object
prototype	String	toString	undefined	valueOf

You should also keep in mind that JavaScript is often used with Java, and so it is very important to avoid using some of the Java reserved words. Similar to windows reserved

words, JavaScript is not limited to html, but can also be used as the programming language for other applications. Lastly, avoiding the use of all HTML event handler is also recommended.

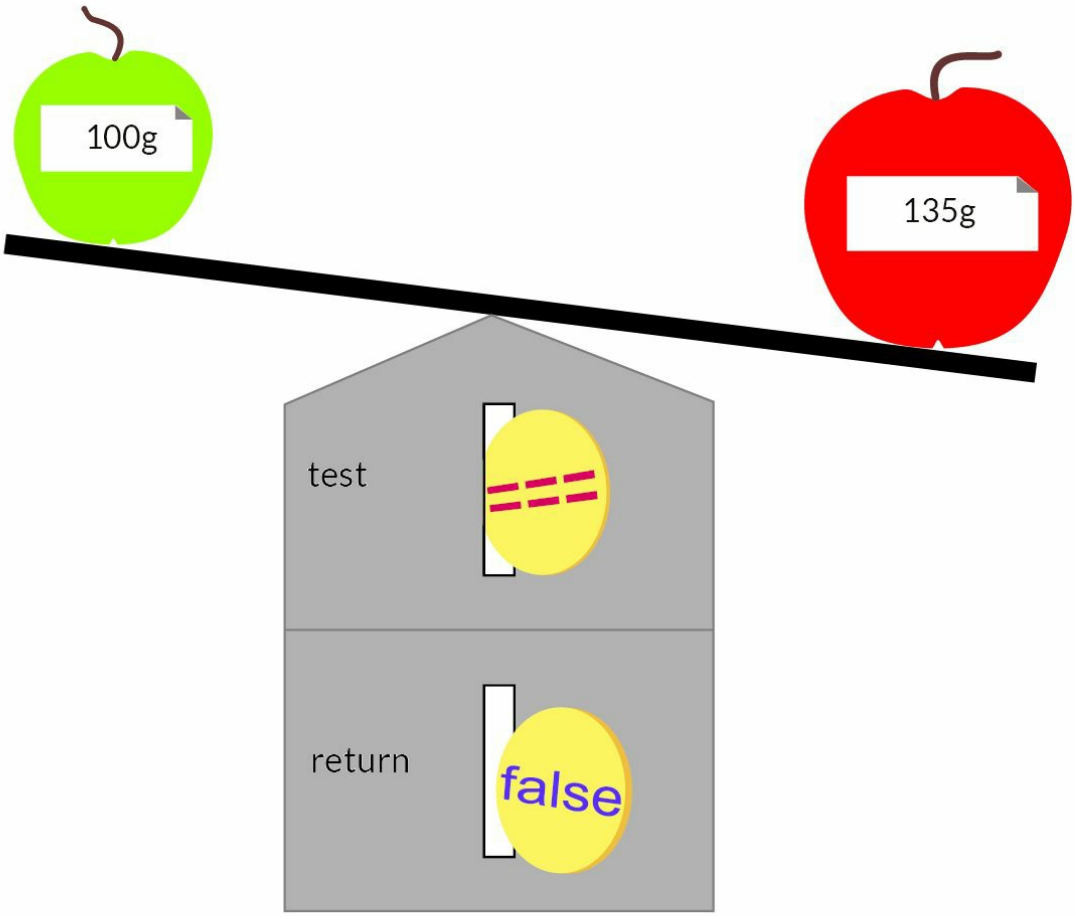
Exercise: 18
Reserved words.

PROBLEM: Find and correct the error

```
1. //declare some variables
2. var default = "1";
3. var export = "Export Number";
4. var modify = default + " " + export;
```

ANSWER: jsvisually.com/18.html

Comparisons.



Comparison operators are used to compare (test) two values and return true or false, depending on the equality based query.

Probably the most widely used comparison operator is the “equal to” (==) operator.
It can be used with different type of variables such as:

- Strings
- Numbers
- Variable with math expressions
- Variable with a variable

When you are comparing strings, then case sensitivity matters! The next most used operator is the opposite of “equal to”, which would be “not equal to” (!==) and shares the same properties as its counterpart.

The following are valid ways to use either == or !== in if statements:

```
1. if (fullName == "John" + " " + "Doe") {
2. if (fullName == firstName + " " + "Doe") {
3. if (fullName == "firstName + " " + lastName) {
4. if (totalCost == 3.14 + 234) {
5. if (totalCost == materialsCost + 678) {
6. if (totalCost == materialsCost + laborCost) {
7. if (x + y == a - b) {
```

Now, === is “equal value and type” while !== is “not equal value or not equal type” and they ensure that you do not compare strings to numbers and so on.

Unlike the previous ones, the following operators are used straightly with numbers:

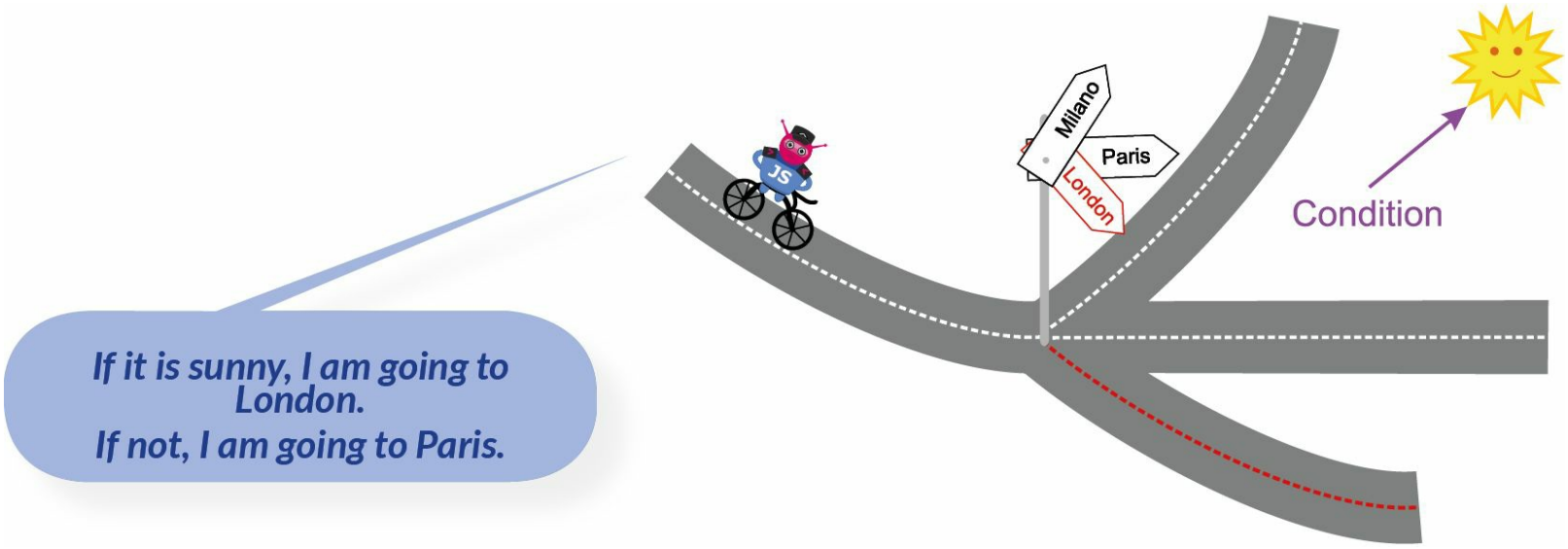
- greater than
- < less than
- >= greater than or equal to
- <= less than or equal to

Exercise: 19
Comparisons.

PROBLEM: Compare the colors of an orange and an apple and display the result in the document

ANSWER: jsvisually.com/19.html

Conditions.



Conditional statements are a set of commands that executes only when a condition is met. They are very useful when you require input from the user, as you will need to check if the user entered the right information, and in case it didn't, then you need to take action. That is done by using *if ... else* statements.

```
1. if (condition) {
2.    statements_1
3. } else {
4.    statements_2
5. }
```

Conditions can be expression that evaluates to either true or false. Having expressions that evaluate to some result or other action on itself will not work. However, you could have a condition that evaluates to something and then use any of the comparison operators to obtain a true or false return.

If you want to test for more than one condition, then you can use logical operators, and also compound the statement to allow for the use of another if statement inside another.

COMPOUNDING:

```
1. if (condition_1)
2.    statement_1
3. [else if (condition_2)
4.    statement_2]
5. ...
6. [else if (condition_n_1)
7.    statement_n_1]
8. [else
9.    statement_n]
```

USING LOGICAL OPERATORS:

```
if (condition) && (condition 2) {
   statements_1
} else {
   statements_2
}
```

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

However, while you could have infinite if statements inside each other as long as you keep the correct, or even use multiple ones one after another, it is still a very inefficient way to check for multiple possibilities. Therefore, another structure called switch was created.

Switch statements work the same as if statements, yet the difference is that they check for multiple values without having to use more of the same structure, which is the downside to using if statements.

The advantage is that with one switch statement, you can have a program to evaluate an expression and check for unlimited number of values which are assigned to a case label. If a match is found, then the program executes the code associated with that case.

The program would first look for a case clause with a matching label having the value of the expression and transfer control to that clause. If none are found, then it will look for a default clause to fall to, if none is found, then nothing inside the simply none of the clauses get executed. It is recommended to have a default clause to handle the cases in which nothing would otherwise get executed and you would have no way to finding out otherwise.

Exercise: 20
Conditions.

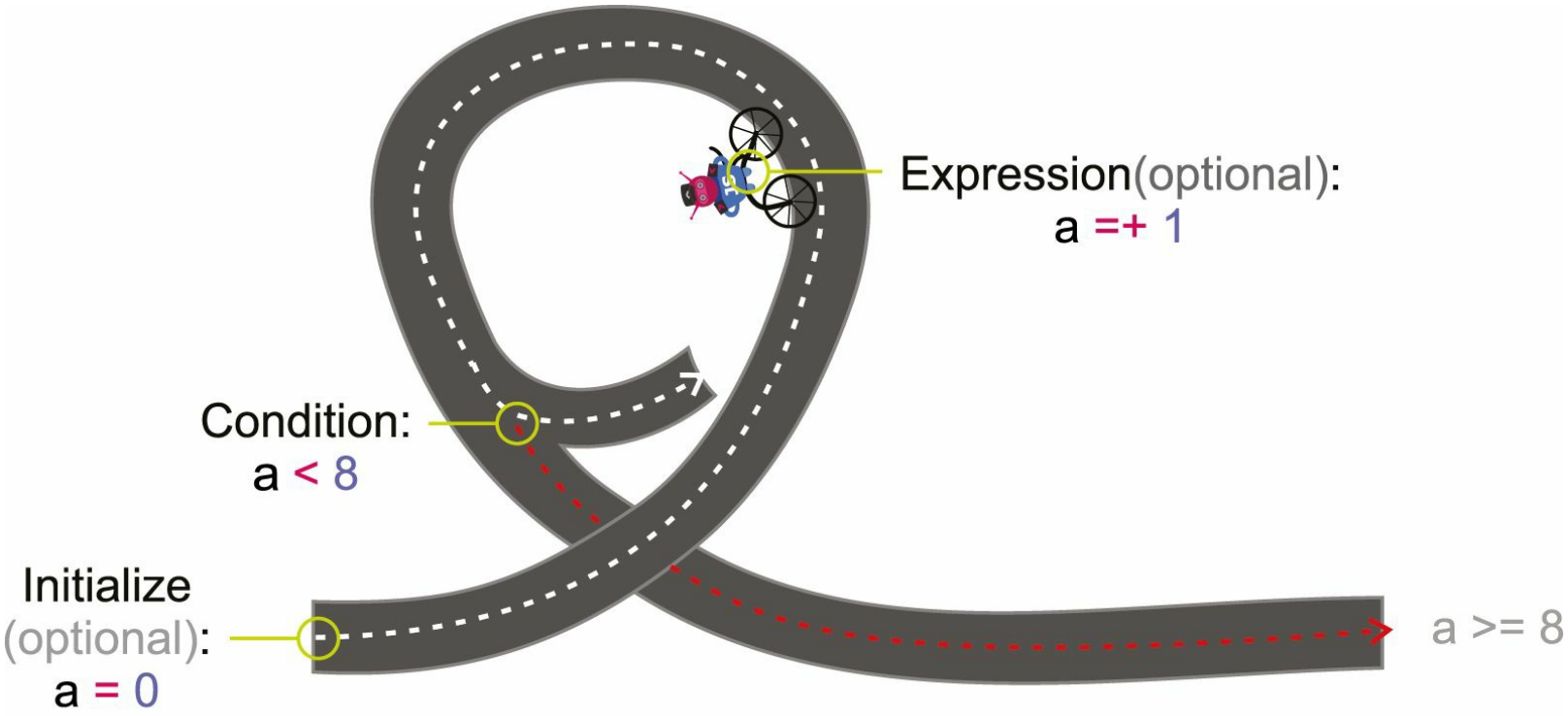
PROBLEM: Private JS wonders what is the distance from Fremont, CA to San Jose, CA?

If it's less than a 30 min drive, he would go visit. Use Google maps and the conditional operator to help him make a decision if he want to go or to stay home.

https://goo.gl/maps/wJwU9

ANSWER: jsvisually.com/20.html

Loops.



Private JS will loop 7 times, then it will proceed on the red path

Loops are very widely used in JavaScript and any other programming language for that matter. The reason is that they are handy at handling repetitive code. If you have to repeat certain blocks of code more than once in your script, then you use functions to modulate that block of code and call it when needed.

However, when you need to repeat certain blocks of code multiple times, one after another, then loops will be the most efficient way.

JavaScript supports two main kinds of loops, which in turn have their own variation:

1. for – It loops through a block of code as many times as specified.

for/in – Used for looping through the properties of an object.

2. while – While a specified condition is met, it will keep looping through a block of code.

do/while – Just like the while loop, loops through a block of code as long as the specified condition is met.

It is an imperative to add some statement to change the values of the variable that is used for condition checking. Otherwise, you will get an infinite loop that will result in freezing, crashing or any other number of malfunctions on the system running the code, as it will run without an end and consume all available memory until the program crashes.

The for loop has the following syntax:

```
for (statement 1; statement 2; statement 3) {
    code block to be executed
}
```

Statement 1 is executed before the loop (the code block) starts.

Statement 2 defines the condition for running the loop (the code block).

Statement 3 is executed each time after the loop (the code block) has been executed.

```
//This loop will run 5 times before stopping
for (i = 0; i < 5; i++) {
    text += "The number is " + i + "<br>";
}

//For/In Loop that goes through the object person
var person = {fname:"John", lname:"Doe", age:25};

var text = "";
var x;
for (x in person) {
    text += person[x];
}
```

The while loop has the following syntax:

```
while (condition) {
    code block to be executed
}
```

In this example the code will loop while the variable i is less than 10.

```
while (i < 10) {
    text += "The number is " + i;
    i++;
}
```

If you do not add the i++ after the main code is executed, then the loop will never end and your browser will crash as mentioned before. With the for loops, it is harder to have this bug, as the increment statement is required. You would have to make it run for a very long time before it runs out of memory, depending on the system.

The difference between a while loop and a do/while loop is that the while loop checks the condition before running the code, and the do/while will instead run the code and then check for the condition before looping.

```
do {
```

```
    text += "The number is " + i;
    i++;
}
while (i < 10);
```

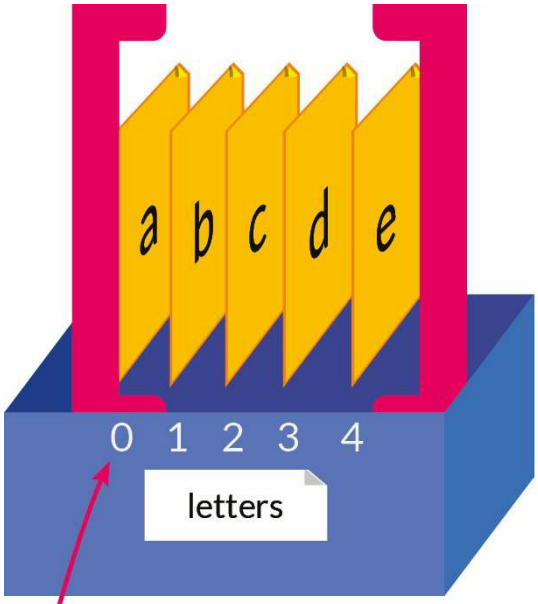
Exercise: 21
Loops.

PROBLEM: Print the numbers 11 to 23 separated by a comma in the document window.

Don't worry about the last comma yet.

ANSWER: jsvisually.com/21.html

Arrays.

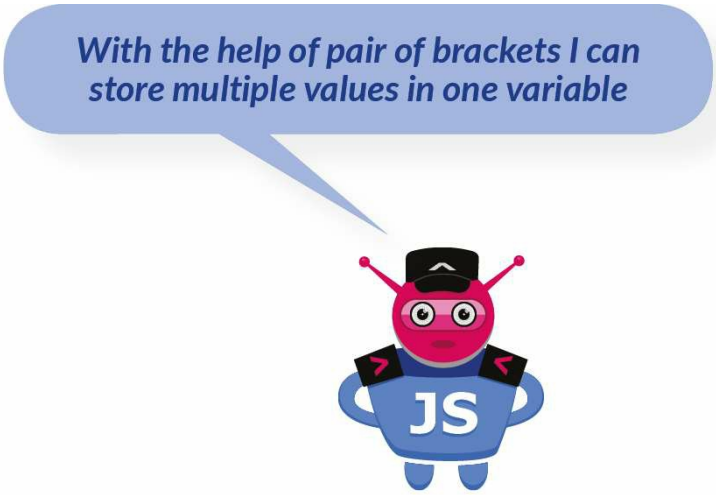


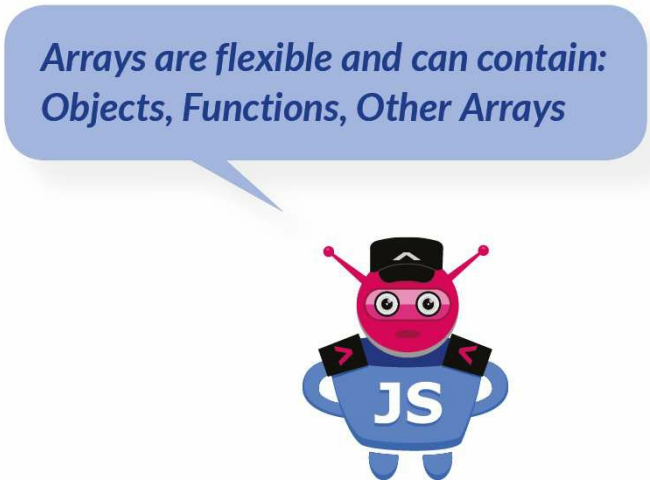
Zero Based Indexing:

[0] is the first element **(a),**

[1] is the second element **(b),**

[2] is the third element **(c)...**





Arrays are a special kind of variable, where a regular variable only has one value assigned to it. The reason they are special is simple – you can assign multiple values to it, instead of making a whole set of variables to create your own mini database.

To define an array all you have to do is encase the values separated by a comma in a bracket.

```
1. do {
2.     text += "The number is " + i;
3.     i++;
4. }
5. while (i < 10);
```

It can contain as many values as you wish and even mix the types if you know what you are doing. However, it is best to keep arrays with the same types of values in them.

Now, let's say that you need to access one of the values from your new array -- all you have to do is use the variable name and the position of the value in the array in the following

way:

```
1. colors[1]
```

This would be the value “Blue” which you can use as a regular variable and do any can of operations with it.

Something to keep in mind is that the first item of the array will always be 0 instead of one. It might be tricky to remember as we are used to start counting form 1 instead of 0.

This means that the array colors have an index of two, and there are three items on the list.

The naming convention for regular variables also applies to arrays. When checking source code, you can usually identify arrays in JavaScript because the name is often a plural, like colors instead of color, and so on. This is due to the fact that arrays are a list of things. Also, just like regular variables, you only declare an array once, and then you can change the values by assigning them by using the following format:

```
1. arrayName[index] = newValue;
```

Note that the keyword var is not used because it would be declaring a new variable instead of changing the value of the current index in the array.

Exercise: 22
Arrays.

PROBLEM: Create an array of 5 numbers (1 to 5)

Change the first element to your name.

Change the last element with your 3 favorite colors.

ANSWER: jsvisually.com/22.html

Arrays methods.

Array objects have predefined methods, and they are the core of array functionalities in JavaScript. These methods will allow you to convert values to string, to add or remove values, to shift and many more data manipulation.

It is easy to remove elements while working with arrays. All you have to do is pop() them as if they were balloons.

```
1. var colors = [“Blue”, “Red”, “Orange”];
2. colors.pop();
```

From that example, you would be removing the last element on the array. If you would like to add a new element to the array, then the array method push() will add the new element to the end of the array.

```
1. colors.push(“Yellow”);
```

This would make our colors array have the colors blue, red, and yellow, instead of just blue and red.

Now, pop() and push() methods work at the end of the array, if you want to work with the beginning of the array, then you can use the array method shift() to pop the first element of the array and then shift all the remaining elements to the left. The equivalent of pushing to the first element is the array method unshift() which would add a new element to the beginning of the array.

A very helpful array method that is often used is sort() which sorts the array alphabetically. However, when you have an array of numbers, then you have to do some extra work to

avoid incorrect sorting, since sort() works with strings, and thus, your numbers will be sorted as string and “30” will be bigger than “20” only because 3 is bigger than 2. The clever way to solve this problem is by providing a compare function inside the sort method.

```
1. var scores = [30, 95, 90, 50, 20, 10];
2. scores.sort(function(a, b){return a-b});
```

That will produce the correct sorting for numbers.

Another widely used array method is reverse() which as the name implies will reverse the elements in an array. A nice trick is to sort the array first and then use the reverse method to have them in a descending order.

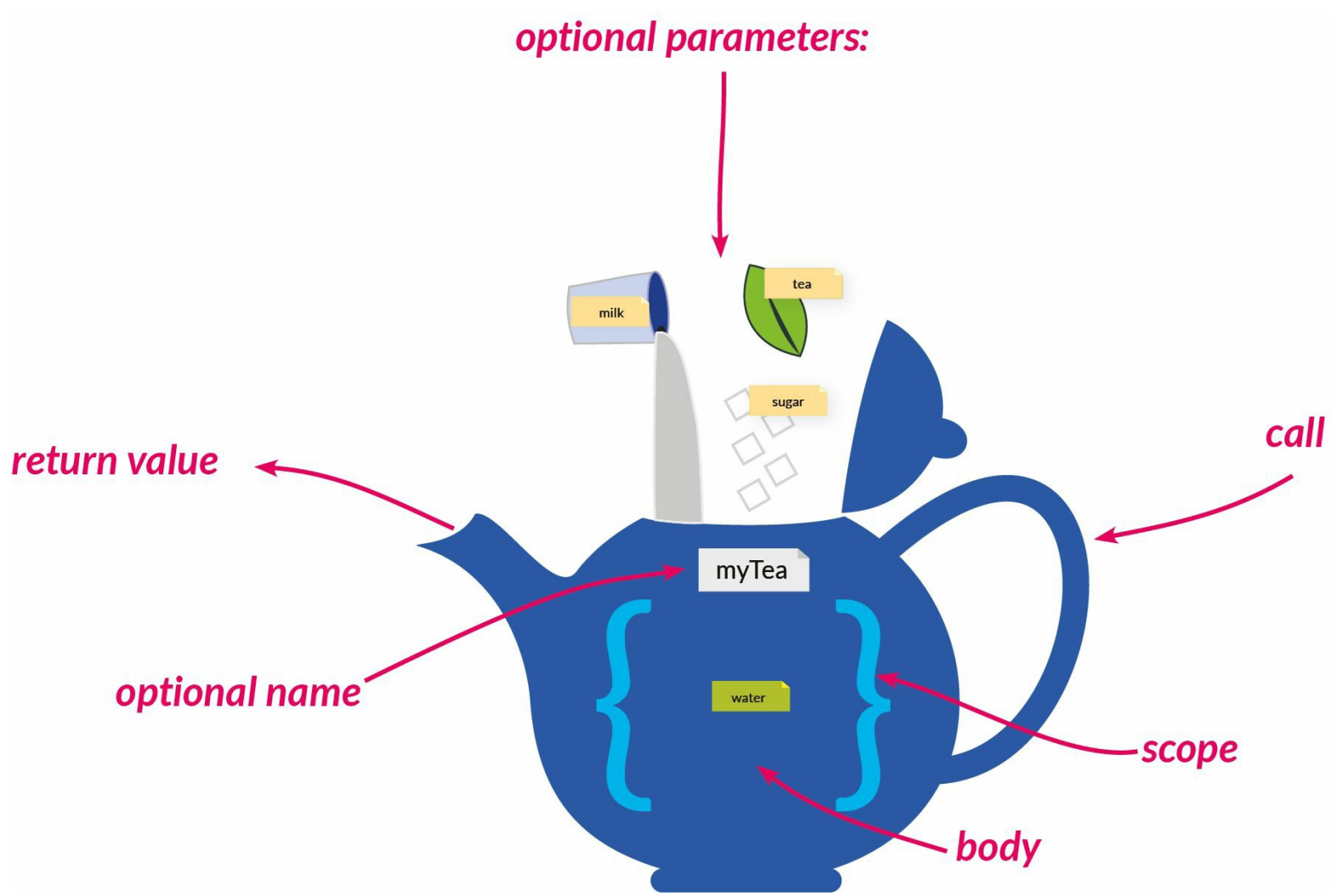
Exercise: 23
Arrays methods.

PROBLEM: Use a loop to create an array containing the numbers 11 to 23.

Display the numbers on the screen

ANSWER: jsvisually.com/23.html

Functions.



Functions (Fun actions) are group of codes that can be called later on and together perform a particular task. They are executed when they are invoked in the program. Basically, they run when you call for them.

Since you can invoke functions as often as you want and pass parameters to them, they are the best way to clean up your code by creating functions out of block of codes that will be used more than once. Even if they are invoked once, they are also a way to organize your code, so you can easily locate the block of code in case you need to modify it instead of scrolling through numerous lines of code trying to find the few lines that you need.

The syntax for a function is the following:

```
1. function myFunction(p1, p2) {
2.     return p1 * p2;          // The function returns the product of p1 and p2
3. }
```

Functions are defined with the function keyword, followed by the function name and parentheses. If you want to add parameters to it, then you put them inside the parentheses separated by a comma, if there is more than one. These parameters are the values of variables that will be used inside the function.

Think of your body as a program and your organs as functions. Yours organs are the

functions, and they communicate by sharing information via variables that are taken as parameter for the organ and they do their own thing and the whole body functions in harmony.

There are three ways to invoke functions:

1. When an event occurs, such as the click of a button by the user.

2. When it is invoked from somewhere in the code.

3. When it is self-invoked.

The return statement in a function is what provides the resulting value to the invoker. That is why you normally assign the result of a function with a return value to a variable.

```
1. var x = myFunction(4, 3);       // Function is called, the return value will end up in x
2. function myFunction(a, b) {
3.     return a * b;                // Function returns the product of a and b
4. }
```

Invoking a function is as easy as writing the function name followed by the () operator. If parameters are required, then they must be added as well. In JavaScript, you can use functions as variables. This means that instead of assigning the function to a variable, to then use that variable, you could just invoke the variable itself and the result will be the same.

Use:

```
1. var text = "The temperature is " + toCelsius(32) + " Centigrade";
```

Instead of:

```
1. var x = toCelsius(32);
2. var text = "The temperature is " + x + " Centigrade";
```

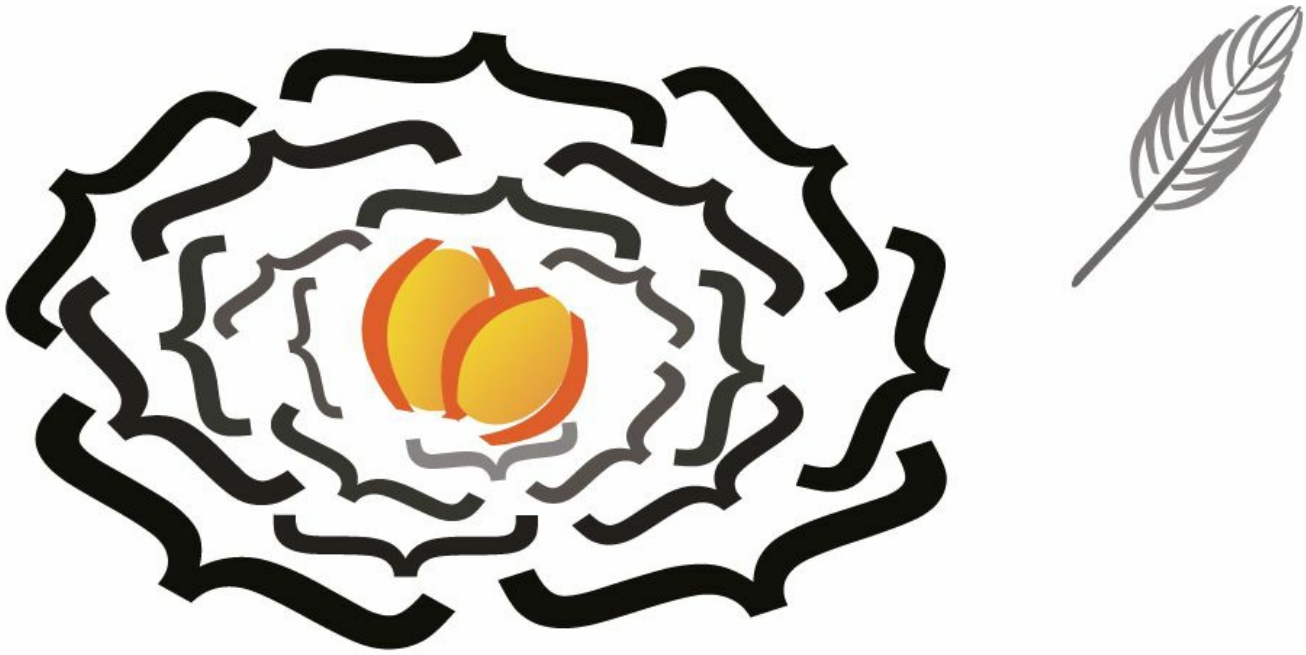
Exercise: 24
Functions.

PROBLEM: Make a function 'cm2in' that converts CM(centimeters) to IN(inches). Round the result with Math.round() and add the inch symbol (") after the value

RESULT: 6"

ANSWER: jsvisually.com/24.html

Nesting.



Avoid deep nesting when possible, because it slows down the script and can be confusing to other programmers

Nesting is the enclosing of code blocks into another. You should avoid deep nesting when possible, because it slows down the script and can be confusing to other programmers. Similar to using many if statements inside another, nesting has the same drawbacks.

This is a pattern for a function inside a normal function:

```
1. function foo(a, b) {
2.     function bar() {
3.         return a + b;
4.     }
5.
6.     return bar();
7. }
8.
9. foo(1, 2);
```

Patterns like that should be avoided. On the other side, the next example is perfectly acceptable:

```
1. function hypotenuse(a, b) {
2.     function square(x) { return x*x; }
3.
4.     return Math.sqrt(square(a) + square(b));
```

For the function hypotenuse(), you get a second function inside called sqrt() from the math library. So each time hypotenuse is called, it will also call the function sqrt(). However, the function foo has to create a function bar for each instance of foo(). One might think that is alright, but this causes unnecessary work for the JavaScript engine and wastes CPU cycles when the function is called multiple times. Because functions do not change, they always do the same thing and it would be best to optimize the code by placing the bar(0 function outside foo() and just have foo() call for it when needed, like the hypotenuse() function does.

```
1. function foo(a, b) {
2.     return bar(a, b);
3. }
4.
5. function bar(a, b) {
6.     return a + b;
7. }
8.
9. foo(1, 2);
```

This new code will be much more efficient, even if the difference is minimal in short runs. The big gains are on big computations. Time resources are saved.

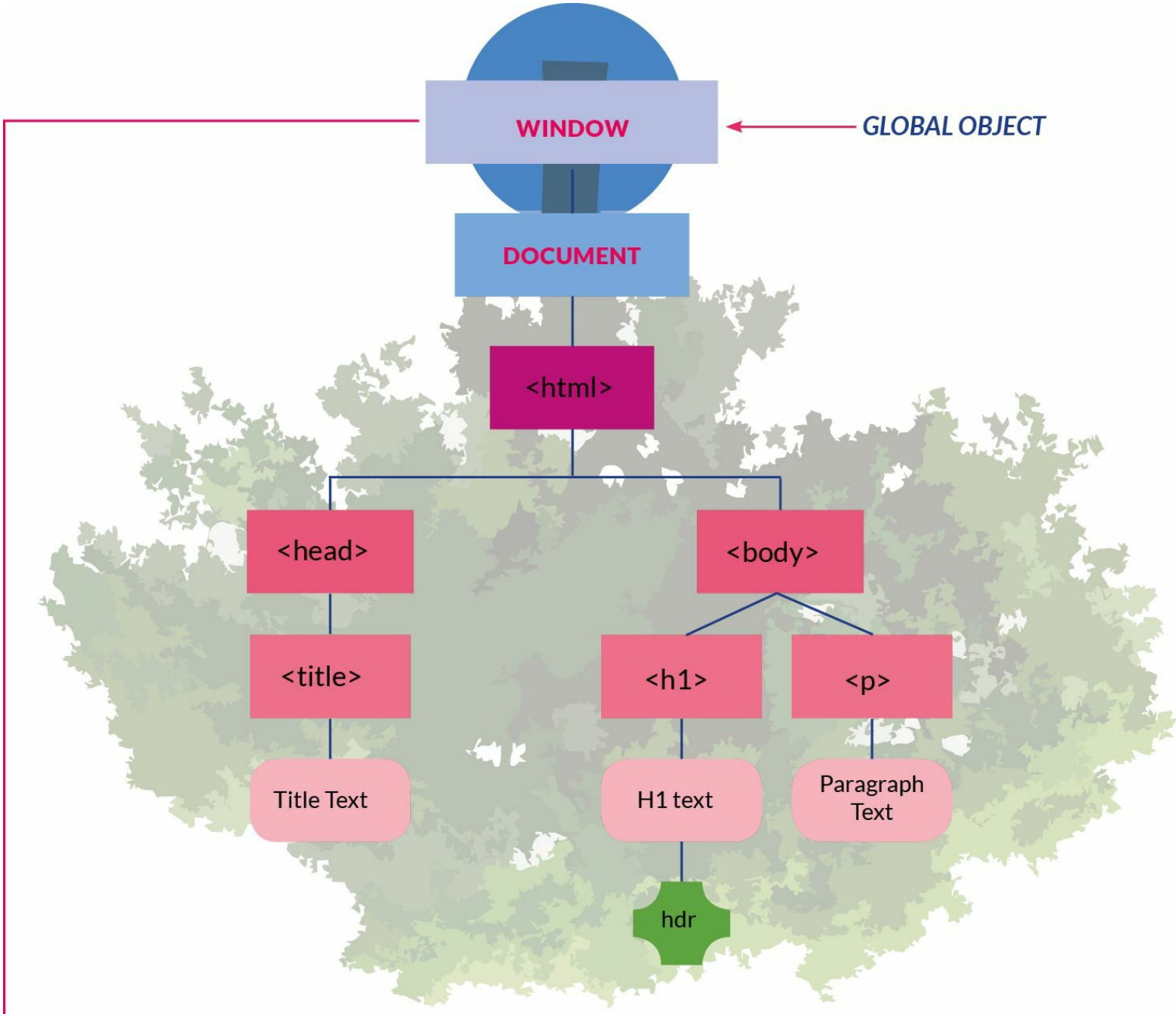
Exercise: 25
Nesting.

PROBLEM: Fix the error in the nested 'if else' statement

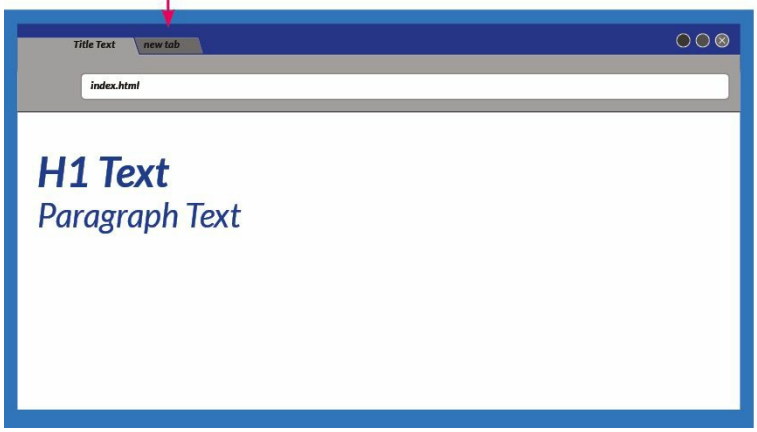
```
1. if (true) {
2.     return 1;
3. } else {
4.     return 0;
5. }
6. Syntax Error
```

ANSWER: jsvisually.com/25.html

The DOM node tree.



Each tab has its own window object



HTML representation of the DOM

```
1 <html>
2     <head>
3         <title>Title Text</title>
4     </head>
5     <body>
6         <h1 id="hdr">H1 Text</h1>
7         <p>Paragraph Text</p>
8     </body>
9 </html>
```

index.html

All HTML elements and attributes are nodes in the Document Object Model (DOM). Take a look at the following HTML code as a starting point:

```
<html>
  <head>
    <title>The title</title>
  </head>
  <body>
    The body
  </body>
</html>
```

The DOM node tree will look like this:

HTML tags are element nodes in the DOM tree, and pieces of text become text nodes. In the graph, you get the HTML node with its two children, HEAD and BODY, from which only HEAD has a child tag.

The idea of the Document Object Model is that every node is an object and its properties can be changed with CSS code. You can also change the content of the nodes by searching the DOM for the target node and create new elements to be inserted on the fly. To be able to do this, you first need to know what the DOM looks like and what it contains. Once you know that, then you can begin with your DOM manipulation. This is something that you will learn in the next chapter.

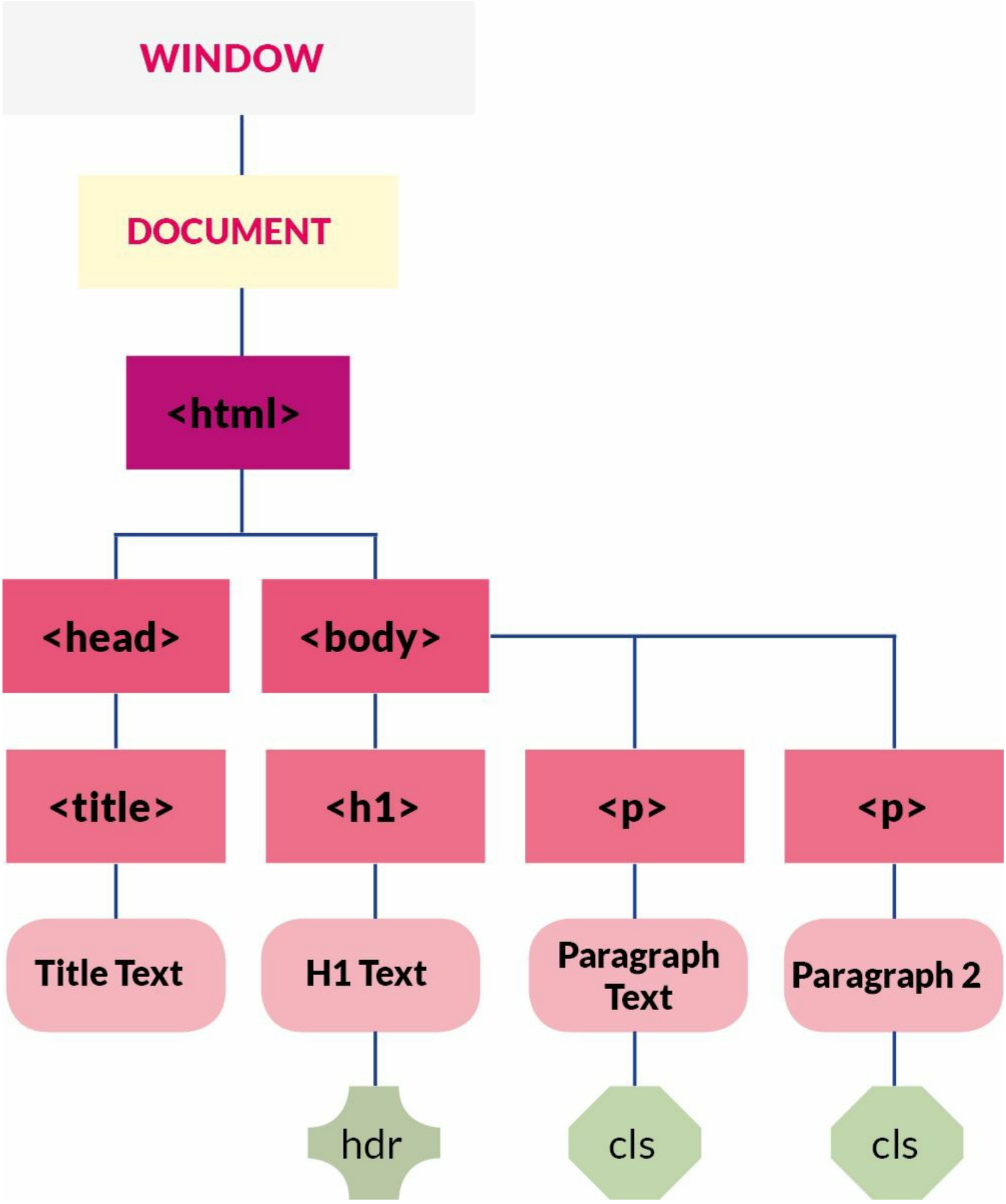
Exercise: 26
The DOM Tree.

PROBLEM: Draw a node tree diagram representing the following HTML

```
<html>
  <head>
        <title>Draw me</title>
  </head>
    <body>
          <h1 id="hdr">Hello</h1>
          <h2>Header tag</h2>
          <p id ="para">My text</p>
          <script src = "my.js"></script>
    </body>
</html>
```

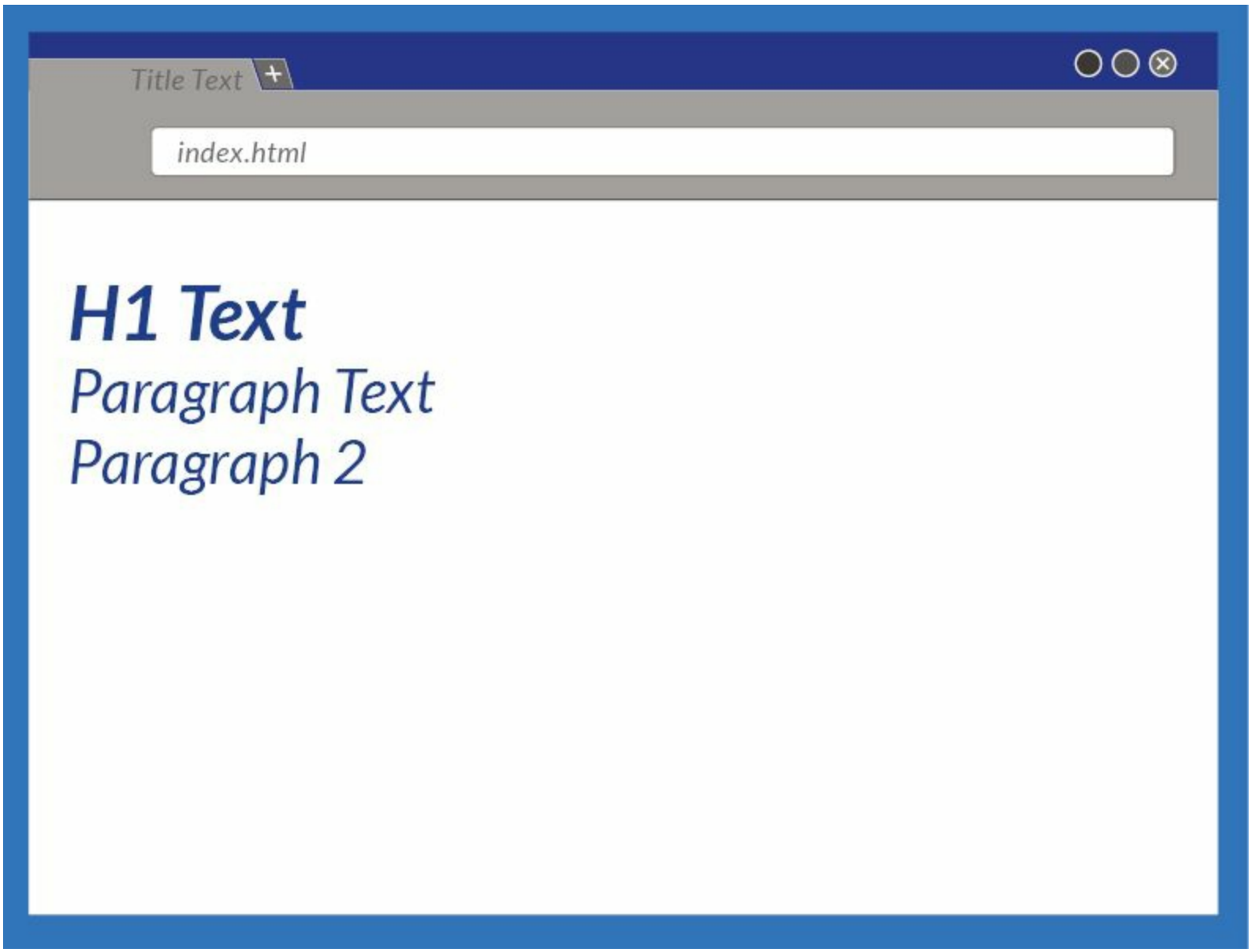
ANSWER: jsvisually.com/26.html

Manipulating the DOM.


WINDOW
DOCUMENT
<html>
<head>
<body>
<title>
<h1>
<p>
<p>
Title Text
H1 Text
Paragraph Text
Paragraph 2
hdr
cls
cls

```
1. <html>
2.   <head>
```

```
    <title>Title Text</title>
  </head>
    <body>
      <h1 id="hdr">H1 Text</h1>
      <p class="cls">Paragraph Text</p>
      <p class="cls">Paragraph 2</p>
      <script src = "main.js"></script>
    </body>
</html>
```



When the web page is loaded, the browser automatically creates a Document Object Model of the page. As you learned before, every element of the page, excluding the document itself, is a child of another element. With the object model, you can use JavaScript to create dynamic HTML. JavaScript can change all the HTML elements, attributes, and CSS styles on the page. Furthermore, adding and removing elements and attributes, and even creating new HTML events on the page is possible.

With the usage of id in your HTML, you have access to specific nodes instead of just all the nodes of a certain type. That is why it is recommended to have different id names, to allow for more precise control.

To work with a particular id, you use the getElementById method, and if you want to interact with the content of the same element, then you use the innerHTML property. That is the easiest way to get to a particular node, and modify it with JavaScript.

```
<html>
<body>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>
</body>
</html>
```

In the previous example, we interacted with the paragraph element using its id by using the getElementById and modified it with the innerHTML property. You can also get element tags with the method document.getElementByTagName(), and the class name with document.getElementByClassName().
That is the advantage of using descriptive names for your methods.

If you want to change HTML elements, then you can use element.attribute= to change the attribute. If you want to set the attribute, then use element.setAttribute(attribute,value). To work the style, you use element.style.property=.

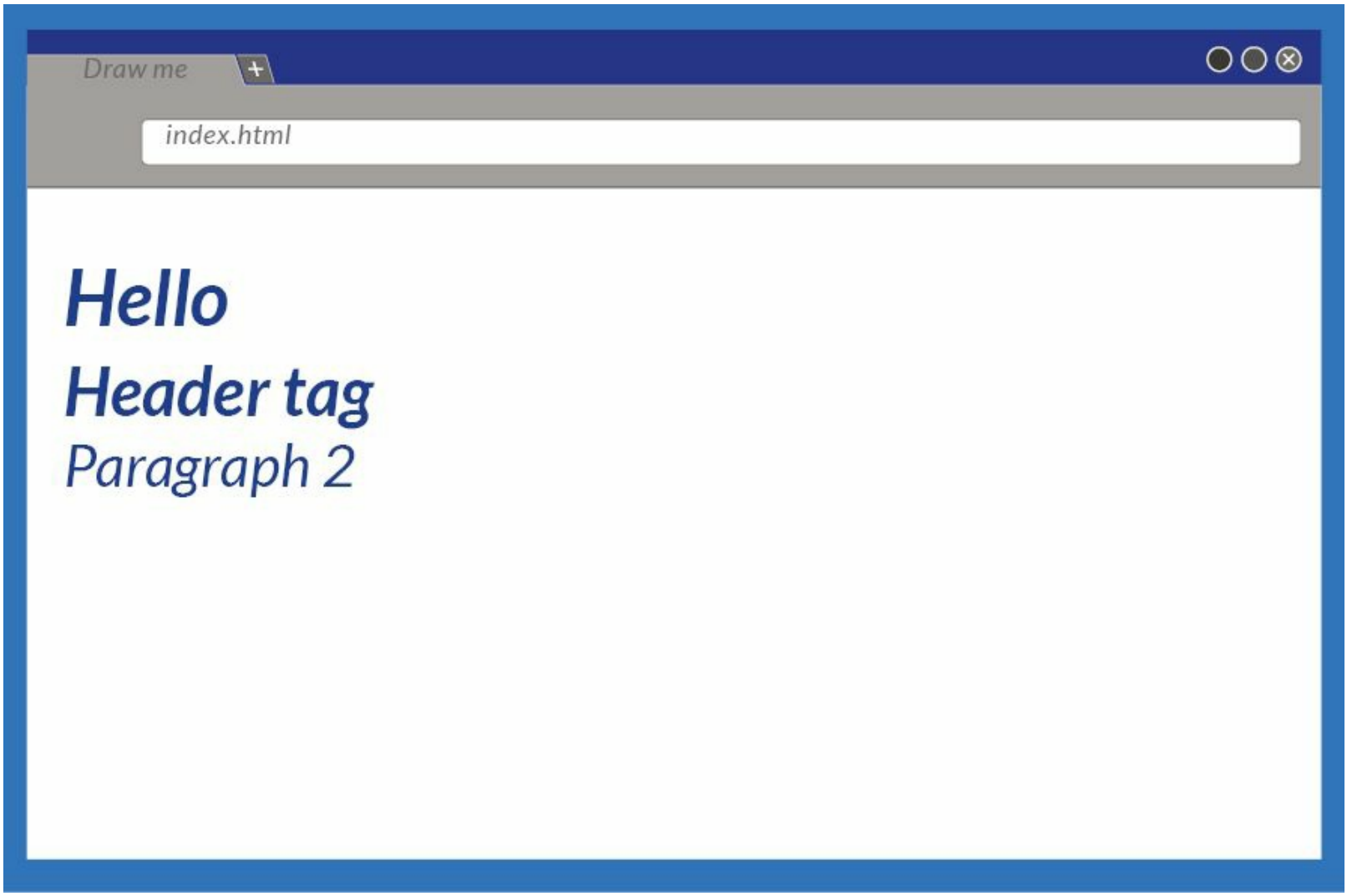
For adding and deleting elements:

- document.createElement() creates an HTML element.
- document.removeChild() removes the child of an element.
- document.appendChild() adds a child to an element.
- document.replaceChild() replaces a child element with another.
- document.write(text) writes into the HTML output stream.
- document.getElementById(id).onclick=function() will add an event handler code to an onclick event.

Exercise: 27
Manipulating the DOM.

PROBLEM: Select the <p> tag with id *"cls"* and change the text to "something else"

```
1. <html>
2.   <head>
3.     <title>Draw me</title>
4.   </head>
5.     <body>
6.       <h1 id="hdr">Hello</h1>
7.       <h2>Header tag</h2>
8.       <p id="cls">Paragraph 2</p>
9.       <script src = "my.js"></script>
10.    </body>
11. </html>
```



ANSWER: jsvisually.com/27.html

Function statements vs function expressions

```
1. //Functions are values
2. //function Statements (Declarations) require a name,
3. //are defined at parse time and
4. //can be called before being declared because of hoisting
5. foo();
6. function foo() { return 8; }
```

there is nothing before the word "function", so it's a statement

this is an expression because of the assignment operator

the grouping operator can only contain an expression

```
1   //function Expression.
2   //name is optional
3   //defined at run time
4   var foo = function foo() { return 8; };
5   foo();       //8
6
7   //function Expression with parameters
8   var boo = function (x, y) { return x + y; };
9   boo(2,3); //5
10
11  //anonymous function Expression
12  var moo = function() { return 1; };
13  moo();   //1
14
15  //this is a function Expression
16  //because it is inside a grouping operator ()
17  ( function doo() { return 8; }() );
18  //8
```

Function statement is where a statement begins with the word “function”. If not, it’s a function expression.

When a function is declared, it is saved for later execution after it is invoked. Since in JavaScript, functions are objects, they have both properties and methods.

```
1. function functionName(parameters) {
2.     code to be executed
3. }
```

While the functionName is mandatory and must follow the naming convention explained in early chapters, the parameters are optional and there can be zero or as many as you need for the function separated by commas. It is important to note that if a function is invoked with missing arguments, the values will be set to undefined.

On the other hand, the syntax for a function expression would be:

```
1. var variableName = function(parameters) {
2.     code to be executed
3. };
```

As you can see, it is very similar, with the main difference being the function name, which can be omitted in a function expression to create an anonymous function. However, if you want to refer to the current function inside the function body, then you need to create a named function expression. The name will be local only to the function body.

```
var math = {
  'factorial': function factorial(n) {
    if (n <= 1)
      return 1;
    return n * factorial(n - 1);
  }
};
```

As you can see, function expressions are more flexible than function statements.

Exercise: 28
Function Statements vs Function Expressions.

PROBLEM: In JavaScript we can raise a number to a power with the Math.pow(x,y) method. X is the base and Y the exponent.

Refactor the following function and make it a function expression:

```
1. //raise the number 2 to the power of 10
2. function power(x) {
3.         //not very nice. Let's rewrite
4.          var raise= x * x * x * x * x * x * x * x * x * x + x;          // (x10)
5.         return raise;
6. }
7. power(2);
8. //1026
```

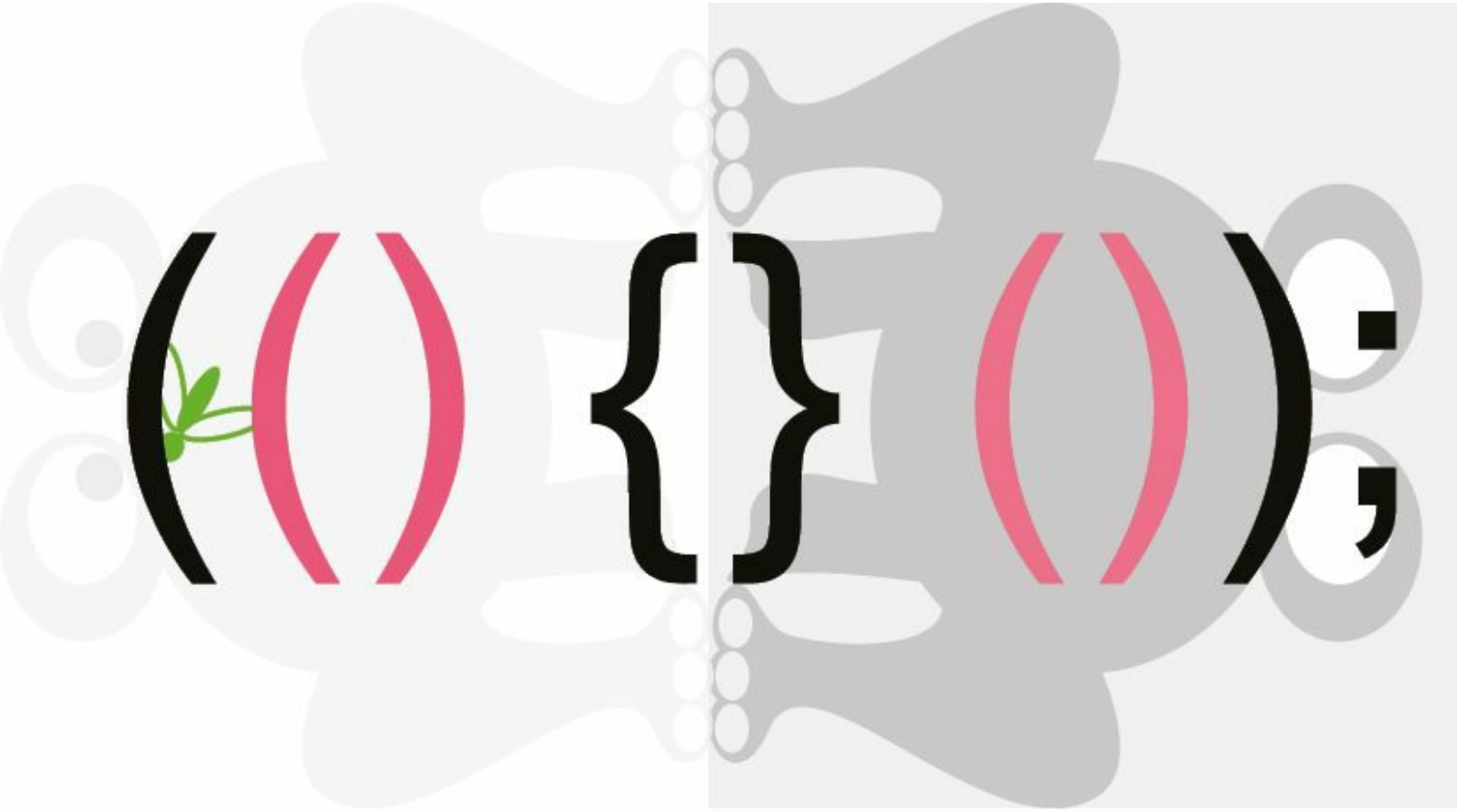
ANSWER: jsvisually.com/28.html

Immediately-invoked function expressions.

```
1.  //we will never call it, so no need for a name
2.  ( function () {
3.          console.log("no need to call me");
4.  }() );
5.   //The second pair of parentheses at the end invokes the function
6.  var x = function() {
7.          console.log("no need to call me");
8.  }();
9.   //The second pair of parentheses at the end calls and closes the function
10. //if we try to call it later, we get an error:
11. x();
12. TypeError: undefined is not a function
```

Anything that will force the interpreter to evaluate whatever comes next, will invoke the function

```
1.  //Any of the following can do the job:
2.  //+, -, ~, !, void, true, &&, new
3.  !function () {
4.          console.log("no need to call me");
5.  }();
6.  //true
7.  +function () { //try to convert to a number
8.          console.log("no need to call me");
9.  }();
10. //NaN
11. 0, function () {
12.         console.log("no need to call me");
13. }();
14. //undefined
```



Remember the mirror frog pattern

Also known as IIFEs, are self-executing functions that are used to avoid variable hoisting and to protect the global space from ‘polluting’. This means that IIFE provides local scoping and this is why popular libraries like JQuery, BackBone.js and others use it to place all library code inside of a local scope.

The syntax for creating a basic IIFE is the following:

```
1. // Anonymous function
2. function(){
3. // my special code
4. }();     // The parentheses make sure the anonymous function gets called immediately
```

Basically, it is an anonymous function without a name, since we will not call it, that is created and invoked immediately via the parentheses at the very end.

With wrap the anonymous function inside the parentheses so the JavaScript parses knows to treat it as a function expression and not as a function declaration. If it is treated as a function declaration, then we will need a name for the function to avoid syntax errors, and it will also need to be invoked at some point. Just to provide a reminder and make the difference between function expression and declaration, check the following:

```
1. Function Expression - var test = function() {};
2. Function Declaration - function test() {};
```

Since the function in our IIFE is a function expression and it is not being assigned to a global variable, no global properties are being created, and thus, all the properties are created locally, so we get local scope.

Another benefit of IIFE is the reduced scope lookup which provides a small performance benefit since we can pass commonly used global objects to an IFFE's anonymous function and then reference them within the IIFE via local scope.

```
1. // Anonymous function that has three arguments
2. function(window, document, $) {
3. // You can now reference the window, document, and jQuery objects in a local scope
4. //The global window, document, and jQuery objects are passed into the anonymous function
5. }(window, document, window.jQuery);
```

Since we are able to pass global objects into the anonymous function as local parameters, we also gain minification optimization, which is another advantage of IIFEs.

One negative thing that IIFE used to have was readability when you have a lot of code inside them. You would have to scroll down all the way if you wanted to find the parameters that you are passing into the IIFE.

The solution is on using a more readable pattern that has become very popular:

```
1. (function (library) {
2.     // Calls the second IIFE and locally passes in the global jQuery, window, and document objects
3.     library(window, document , window.jQuery);
4. }
5. // Locally scoped parameters
6. (function (window, document, $) {
7. // Library code goes here
8. }));
```

This IIFE pattern allows you to see what global objects are being passed into the IIFE without having to scroll down all the way to the bottom of the file.

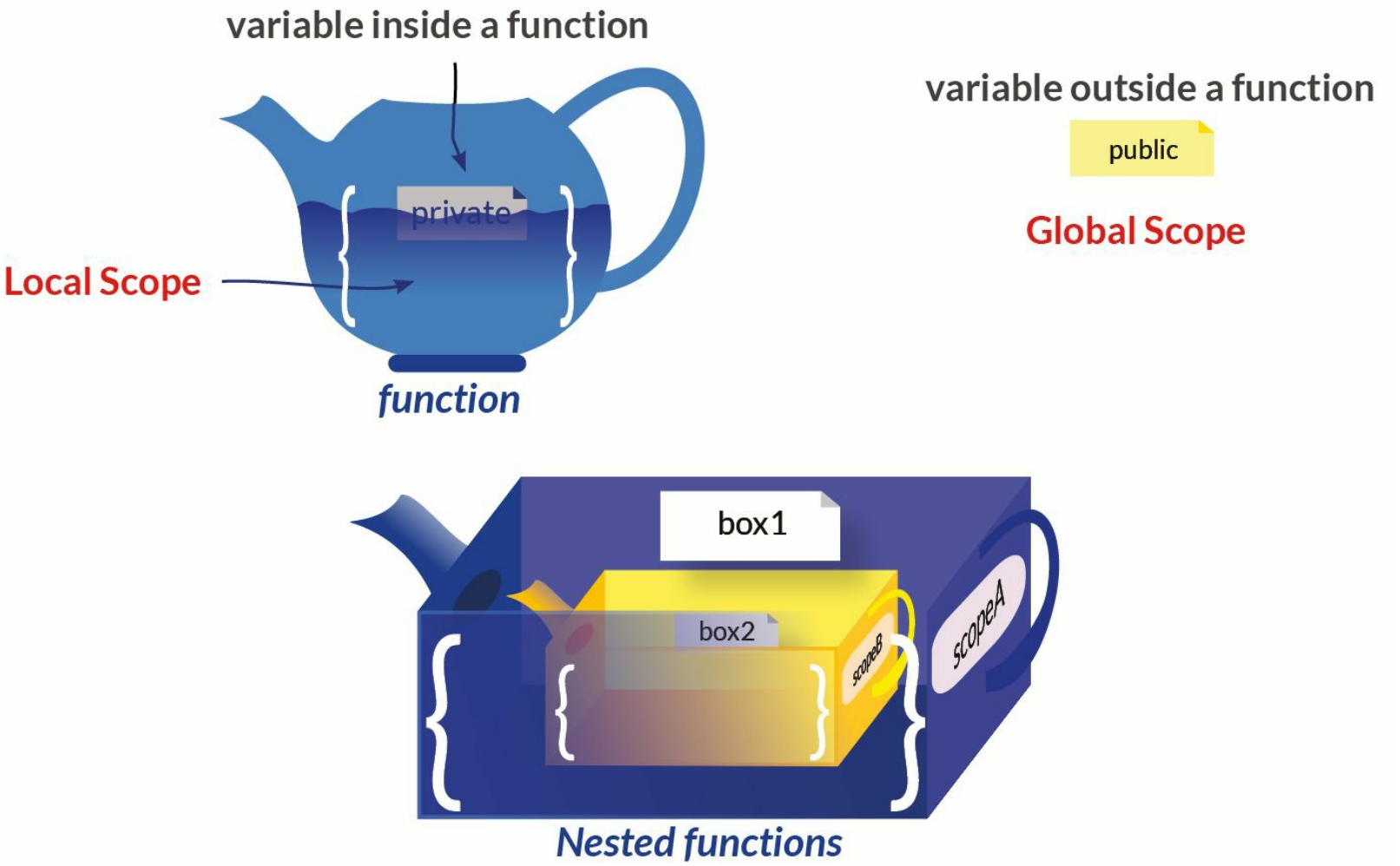
Exercise: 29
IIFE.

PROBLEM: Add no more than 4 characters to make this function self-executing:

```
1. function selfExecute() {
2.         console.log("Make me IIFE, please!");
3. }
```

ANSWER: jsvisually.com/29.html

Scope.



You have seen the word scope being used widely in this book, and if you have not figured its meaning by now, then know this: JavaScript has function-level scope. The variables inside a function are invisible to the outside. The local scope can “see” the global scope, but the global scope can’t “see” the local scope.

In JavaScript, scope is the set of variables, objects, and functions that you have access to. Furthermore, objects and functions are also variables in JavaScript.

Since local variables have a local scope, you can only access them within the function:

```
// code here cannot use carName
function myFunction() {
    var carName = "Toyota";
    // code here can use carName
}
```

However, since local variables are recognized only within their local scope, you could have variables with the same name used in other functions.

Here we have the same function, but with a global variable:

```
var carName = " Volvo";
// code here can use carName
function myFunction() {
    // code here can use carName
}
```

Something to keep an eye for is the automatically global variables. When you assign a value to a variable that has not been declared, it will automatically become global.

```
// code here can use carName
function myFunction() {
    carName = "Volvo";
    // code here can use carName
}x
```

The life cycle of a variable is also a part of its scope. JavaScript variables start when it is declared. Local variables are deleted when the function is completed, while global ones are deleted only when the program or page is closed.

```
// code here can use window.carName
function myFunction() {
    carName = "Volvo";
}
```

Exercise: 30
Scope.

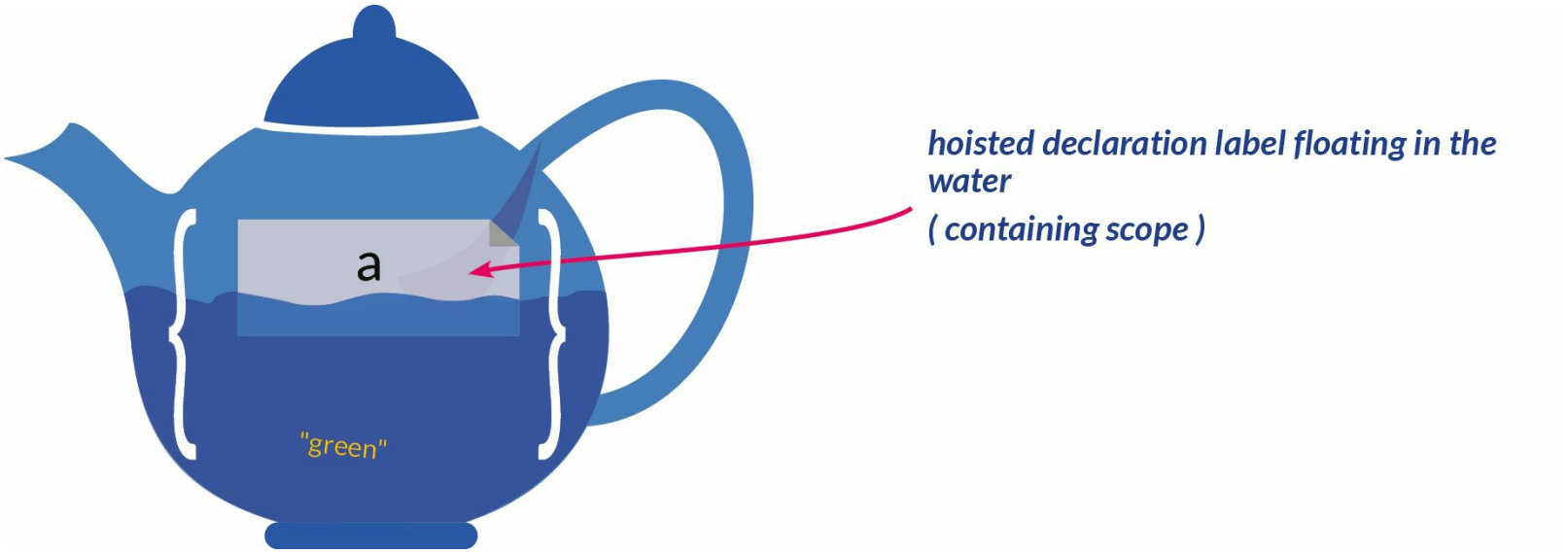
PROBLEM: The function below is suppose to count 10 apples.

Fix the problem!

```
1. var i = "red";
2. function countApples() {
3.         for (i; i <= 10; i++) {
4.               console.log(i);
5.         }
6. }
7. countApples();
```

ANSWER: jsvisually.com/30.html

Hoisting.



Hoisting is when function and variable declarations are moved to the top of their scope. This is the default behavior of JavaScript.

In JavaScript, a variable can either be declared after it has been used, or used before it has been declared.

The following two examples produce the same result:

Example A:

```
x = 5;      // Assign 5 to x
elem = document.getElementById("demo");      // Find an element
elem.innerHTML = x;                  // Display x in the element

var x; // Declare x
```

Example B:

```
var x;          // Declare x
x = 5;          // Assign 5 to x

elem = document.getElementById("demo");      // Find an element
elem.innerHTML = x;                  // Display x in the element
```

JavaScript only hoists declarations, not initializations. Therefore, you must initialize your variables before using them.

```
var x = 5;      // Initialize x

elem = document.getElementById("demo");      // Find an element
elem.innerHTML = x + " " + y;          // Display x and y
```

```
5.
6. var y = 7;     // Initialize y
```

In the previous example, y was declared before being used because of hoisting. Yet, it will still be undefined because initializations are not hoisted.

Since not many developers are familiar with hoisting in JavaScript, it is recommended to declare all variables at the beginning of every scope. This good practice will avoid many bugs, and since it is the way the JavaScript interprets the code, it is a good guideline.

Exercise: 31
Hoisting.

PROBLEM: What is the result of the following code ? Why ?

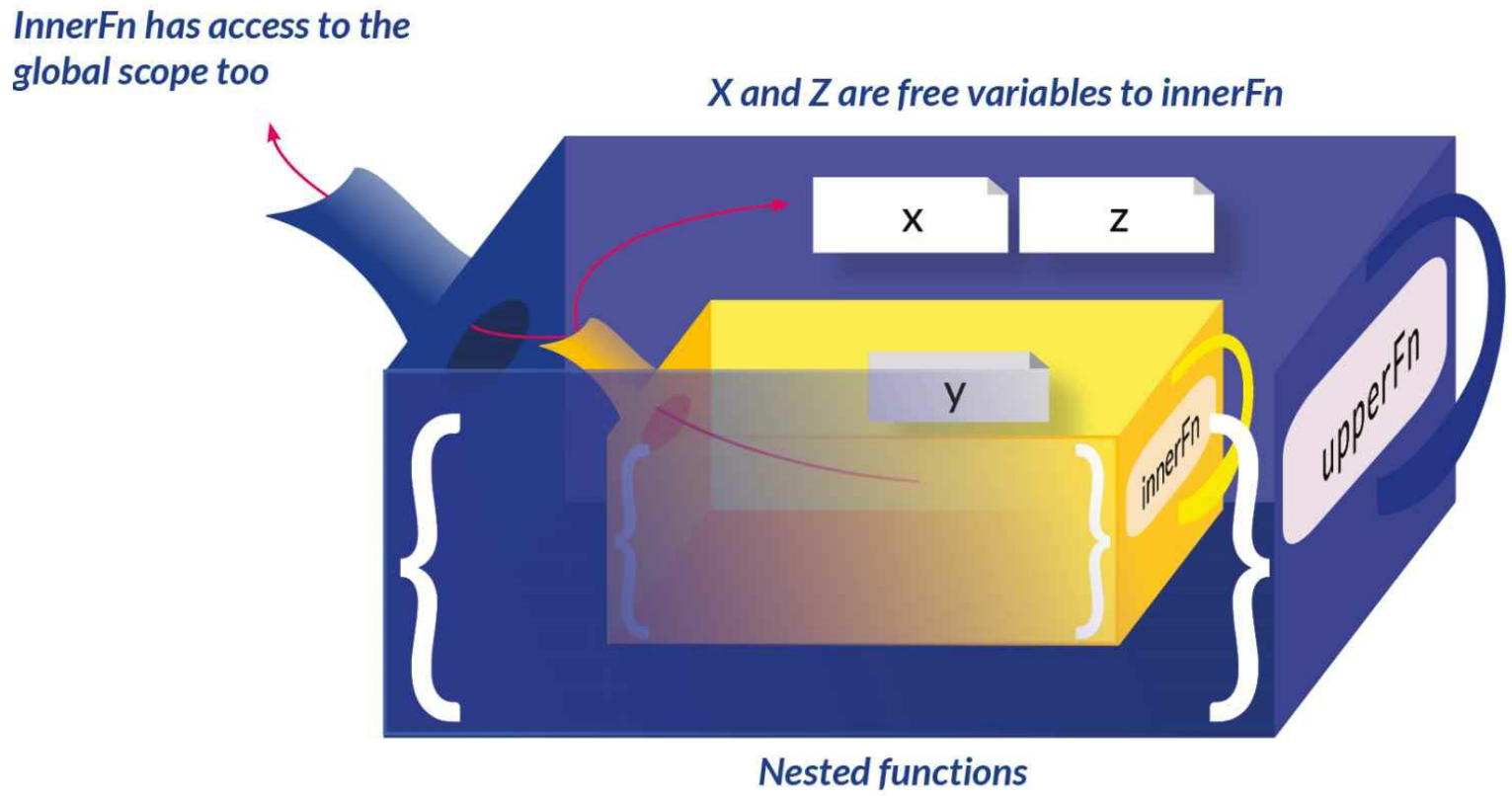
Try to answer without running the code.

```
1. test();
2. var test = function () {
3.         return 1;
4. };
```

ANSWER: jsvisually.com/31.html

Closures.

innerFn remembers the environment in which it was created



Variable 'z' survived the upper function execution

```
1   function upperFn(x) {
2     var z = 8;    //local variable
3
4   //closure is formed during function instantiation, not during invocation
5         function innerFn(y) {          //closure
6                 console.log(x + y + z);
7       }
8         innerFn(3);                    //y = 3
9   }
10  upperFn(2);                          //x = 2
11
12  //13
13  //When 'upperFn' is called, function 'innerFn'
14  //is instantiated and has access to variable 'z'
```

We have seen local and global scope in JavaScript variables. Yet, there is another type of scope, we can make private variables with closures. A closure is an inner function that has access to the variables of the outer function. It has three scope chains:

1. Access to its own scope, variables defined between its curly brackets.

2. Access to the outer function's variables.

3. Access to the global variables.

```
function showName (firstName, lastName) {
var nameIntro = "Your name is ";
// this inner function has access to the outer function's variables, including the parameter
function makeFullName () {
return nameIntro + firstName + " " + lastName;
}
return makeFullName ()
}
showName ("John", "Doe");      // Your name is John Doe
```

One of the most important features with closures is that the inner function still has access to the outer function' variable, even after the outer function has returned. Normally, after a function has returned, its life cycle has ended. Because with closures that is not the case, you can call the inner function later in your program, even if the outer one has already returned.

Closures store references to the variables in the outer function; they do not store the actual values. Because closures have access to update the values of the outer function's variables, they can lead you to bugs when the variable changes due to for loops. To avoid this problem, you can simply use an IIFE.

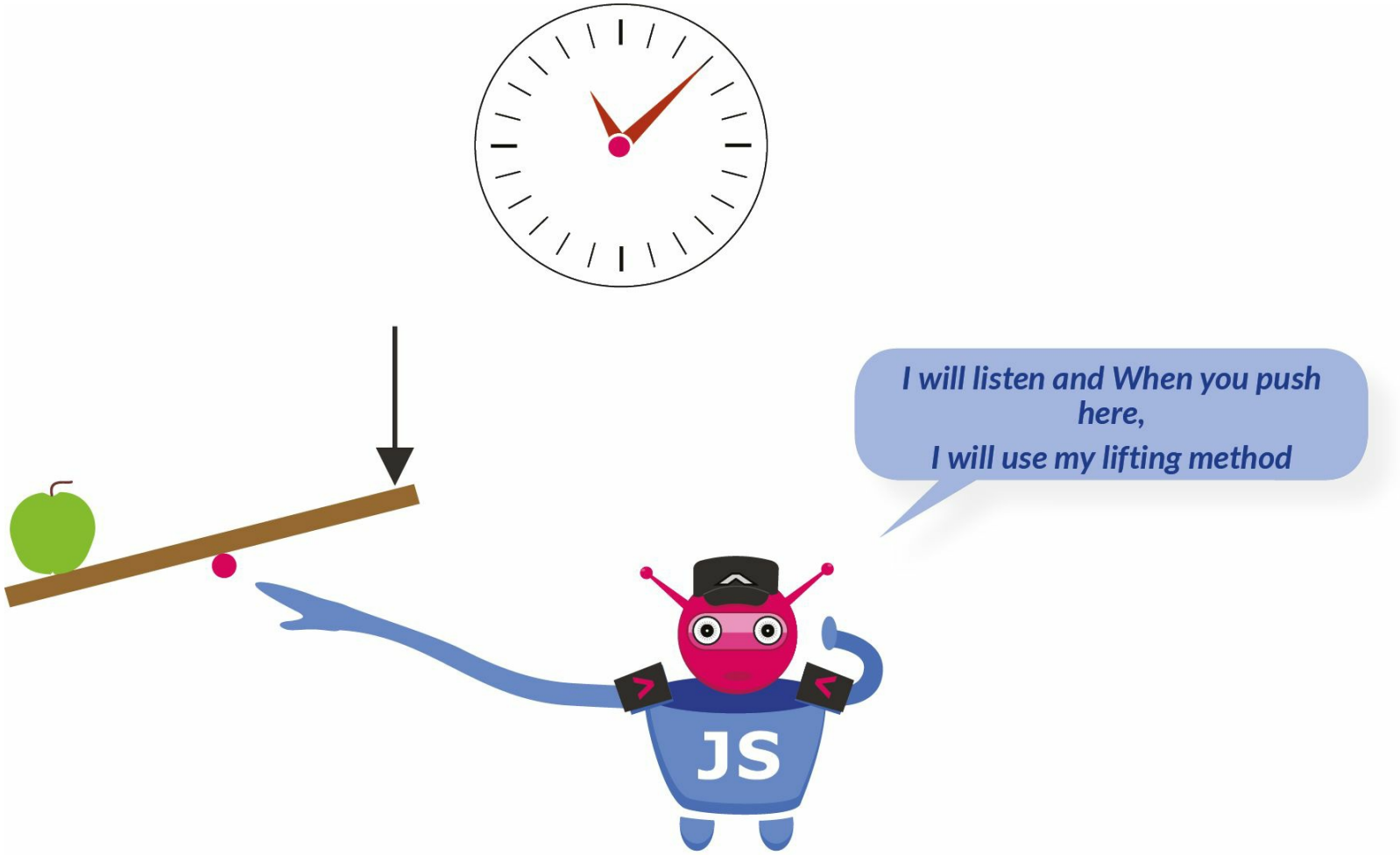
Exercise: 32
Closures.

PROBLEM: Make a function with parameter x that returns a function with parameter y which returns the sum of x and y.

console.log the sum of 3 and 5

ANSWER: jsvisually.com/32.html

Events.




```
element.event = function;
```

Events is what JavaScript uses to make web pages interactive – when you click a button, hover over a picture and have it changed, or any other form of interactivity. Events are actions that happen to HTML elements as a direct result of JavaScript reaction to events.

HTML events can be actions that the browser or the user takes, they could be a web page finishing loading, the change of an input field, and very often the clicking of the mouse. HTML allows event handler attributes with JavaScript code added to it.

```
1. <some-HTML-element some-event='some JavaScript'>
2. <some-HTML-element some-event="some JavaScript">
```

You can use either single or double quotes.

One way to make hyperlinks behave differently than usual is with event handlers:

```
1. <a href="#" onClick="alert('Hi');">Click</a>
```

This will display an alert instead of taking the user to another place.

To interact with buttons we use the "onClick" handler.

```
1. <button onclick='getElementById("demo").innerHTML=Date()'>The time is?</button>
```

This will display an alert instead of taking the user to another place.

To interact with buttons we use the "onClick" handler.

```
1. <button onclick='getElementById("demo").innerHTML=Date()'>The time is?</button>
```

That code will change the content of the element with id=”demo”. If you would prefer to change the content of its own element, then you would use this.innerHTML instead.

```
1. <button onclick="this.innerHTML=Date()">The time is?</button>
```

You can also use an image to trigger these changes, instead of creating a button.

```
1. <img src="button-greet.png" onClick="alert('Hello world!');">
```

The most common HTML events are the following:

- onchange: An HTML element has been changed.
- onclick: The user clicks an HTML element.
- onmouover: When the user moves the mouse over an HTML element.
- onmouseout: When the user moves the mouse away from an HTML element.
- onkeydown: The user pushes a keyboard key.
- onload: The browser is done loading the page.

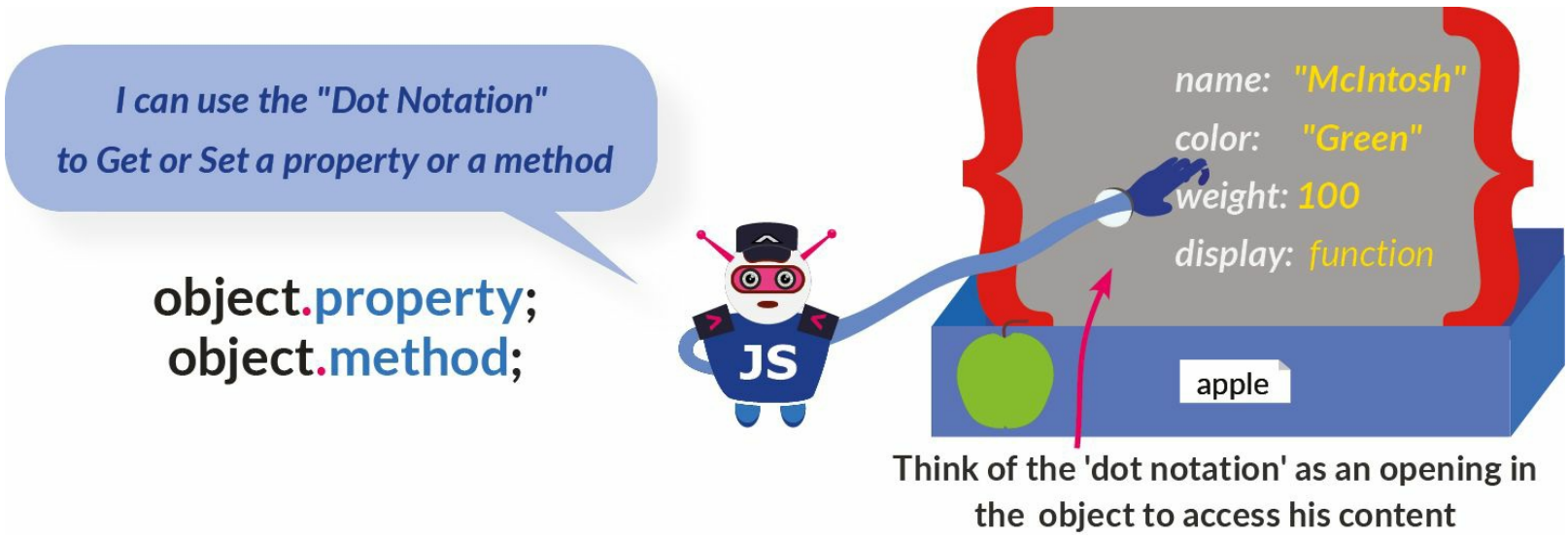
Exercise: 33
Events.

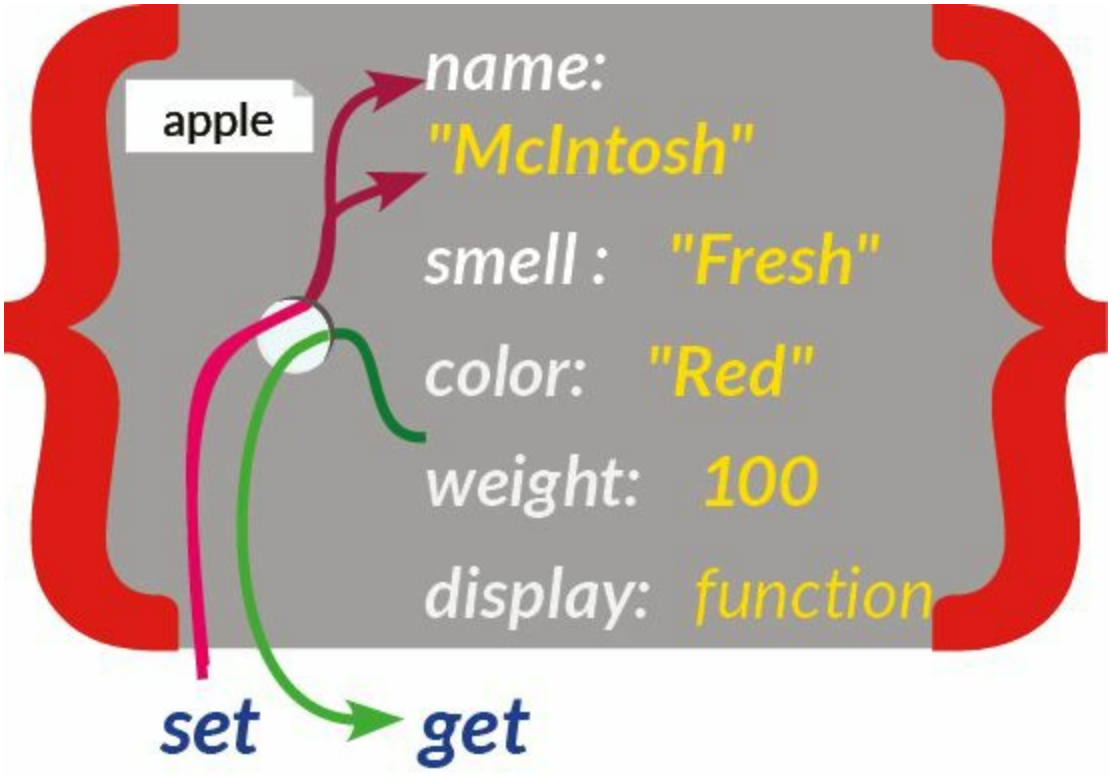
PROBLEM: Create an event listener that calculates a button clicks and displays that number on the screen

```
<html>
  <head>
        <title>Click me</title>
      <p id="hdr">0</p>
      <button onclick = "countClicks();">Click!</button>
  </head>
        <body>
          <script src = "my.js"></script>
        </body>
</html>
```

ANSWER: jsvisually.com/33.html

Objects.






```
1.  //create an empty object
2.  var apple = {};
3.  //create an object and add properties to it
4.  var apple = {
5.          name: "McIntosh",
6.          color: "Green",
7.          weight: 100,
8.          //add method
9.          display: function () { alert("I weigh " + this.weight + "g");}
10. };
```

set

```
//property change with "." dot notation
apple.color = "Red";
//add the new smell property
apple.smell = "Fresh";
```

get

```
//call the method of the apple object
apple.display();
//result: I weigh 100 g
//get the value of a property
var getValue = apple.weight;
```

It is said that when you understand Objects in JavaScript, you understand JavaScript. This is due to the fact that almost everything in JavaScript is an object. Primitive data such as numbers, Booleans, and strings can be treated as objects. Furthermore, dates, math, regular expressions, arrays, and functions are always objects. This might sound like everything is an object, yet, it is important to know that primitive values are not objects -- they can just be treated as objects. Objects are a collection of properties and methods.

While you could use variables to hold information instead of objects, it is very impractical and in some cases extremely difficult to keep all the information linked together, particularly if they are of different types. For example, you want to create an object apple that will have the type of apple, the color, and the weight. All these properties can be stored in variables, but as you have more objects and more complicated ones like a car object for an assembly line, all those variables might get mixed up and it will be difficult to keep track of them. If you treat them as an object type, it will have its own function, properties, and methods all easily available and easier to read from the source code.

```
//create an empty object
var apple = {};
//create an object and add properties to it
var apple= {
        name: “Machintosh”,
        color: “Green”,
        weight: 100,
        //add method
        display:function(){alert(“I weight”+this.weight+”g”);}
};
```

That is a simple apple object. If you need to change its values, you would create getters and setters. As the name suggests, one is to set values, and another is to get values. This is also a security measure as you do not want outside code to have direct access to the properties of the object. Instead, you create doors to regular the information coming in and out securely.

Setter:

```
//property change with "." Dot notation.
apple.color = "Red";
//add the new smell property
apple.smell = "Fresh";
```

Getter:

```
//call the method of the apple object
apple.display();
//result: I weight 100g
//get the value of a propertu
var getValue = apple.weight;
```

- Define and create a single object using an object literal. By using the object literal, you define and create an object at the same time.

As you have noticed from the apple example, an object is like a variable, but it can contain multiple variables with the following format: name : value. The collections of variables are unordered and they are called named values.

There are different ways to create Objects in JavaScript:

```
var person = {
        firstName:"John",
        lastName:"Doe",
        age:50,
        eyeColor:"blue"
};
```

- Define and create a single object with the keyword new. This does the same as the object literal, but the object literal is recommended.

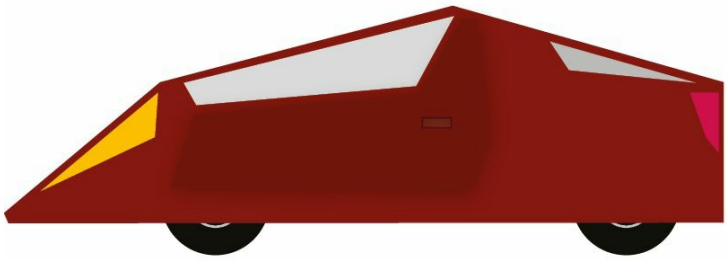
```
var person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

- Define and object constructor, and then create objects of the constructed type. The previous examples are limited compared to this, as they can only create one object instead of being used as a template to make multiple objects with similar properties.

```
function person(first, last, age, eyecolor) {
        this.firstName = first;
        this.lastName = last;
        this.age = age;
        this.eyeColor = eyecolor;
}
var myFather = new person("John", "Doe", 50, "blue");
var myMother = new person("Sally", "Rally", 48, "green");
```

Exercise: 34
Objects.

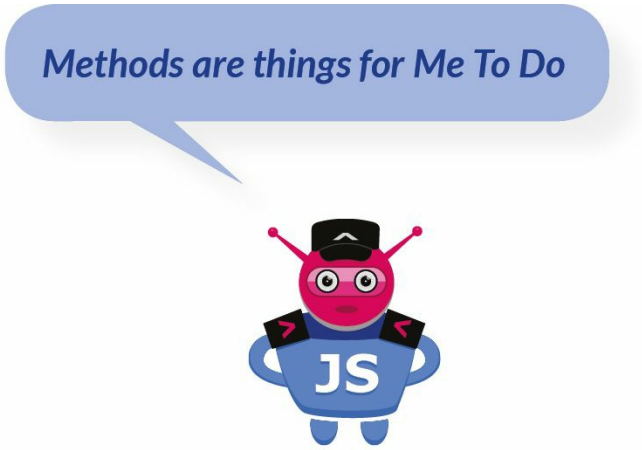
PROBLEM: Create an object literal containing the details of this vehicle's registration form. Delete Year of Manufacture and display the object in the console



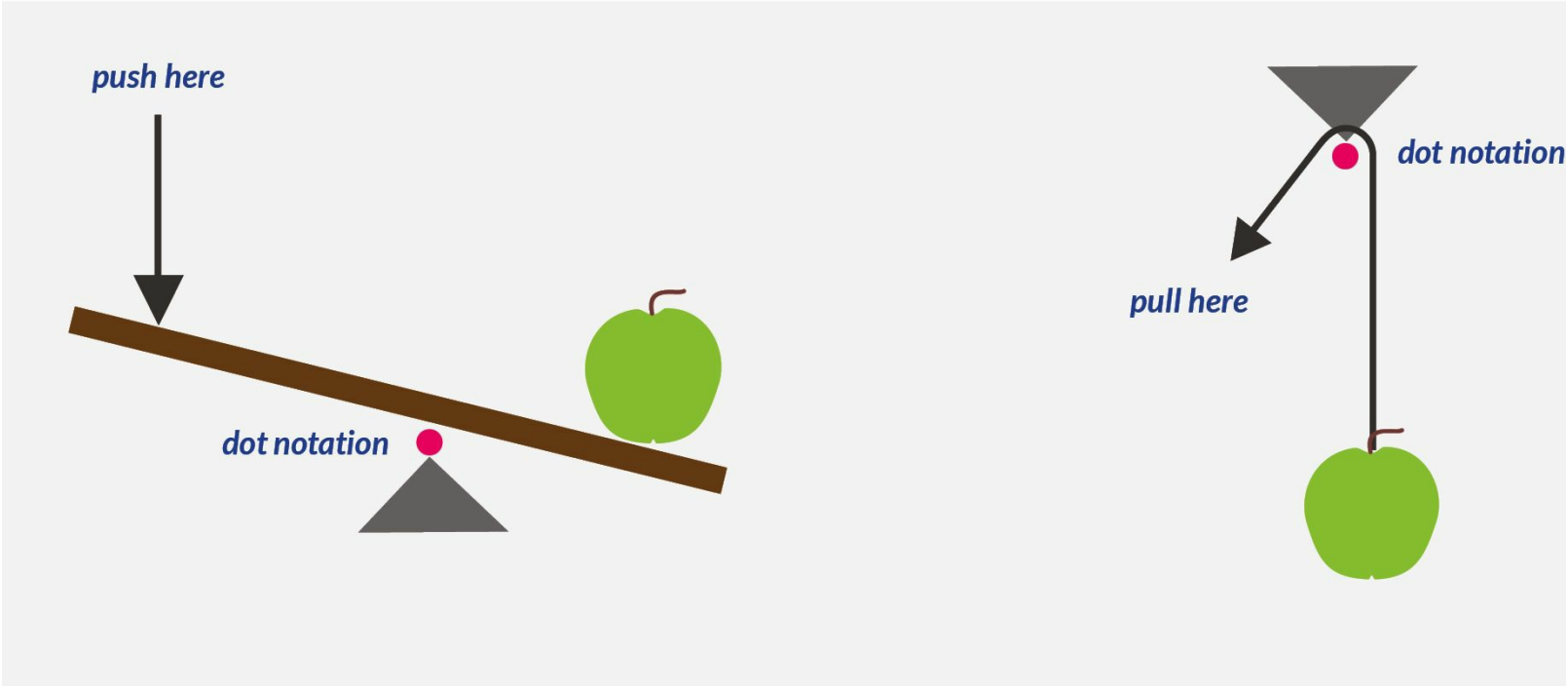
Owners Name:	*John Smith*
Chasis No.:	*451265-458*
Engine No.:	*6565-5656-98*
Make Name:	*Stealth*
Registration Date:	*1 July 2019*
Vehicle Price:	*$145 000*
Color:	*Burgundy*
Year of Manufacture:	*2018*

ANSWER: jsvisually.com/34.html

Methods.



Methods to lift an object



Object methods are object properties containing function definitions, which in turn create actions to be performed on the object. You can create an object method by using the following syntax:

```
1. methodName : function() { code lines }
```

To access the created object method, you use the following syntax:

```
1. objectName.methodName()
```

It is important to remember the parentheses when trying to access an object method. Otherwise, you will get a function definitions.

You can use built-in methods in your objects. For example, if you are working with strings, then you can use the build in string methods to manipulate the data just as it if were a variable.

```
var message = "Hello world!";
var x = message.toUpperCase();
```

Looking back at the person object that we created in the previous chapter, we could define a new method to the object by editing the constructor function.

```
function person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
    this.changeName = function (name) {
        this.last = name;
    }
}
```

Now we have a changeName() method that assigns the value of a name to the person's last property.

```
myMother.changeName("Doe");
```

Another important part of objects is the "this" keyword. The value of "this" when using in a function is the object that the function belongs to. However, when "this" is used in an object, it is the object itself. "this" is a keyword, not a variable, and so its value cannot be changed.

Take a look at the object constructor example: instead of using person.firstName, you can use "this" which belongs to the person function. When it is used on the global scope, then it will not be the person function but the function or object enclosing it.

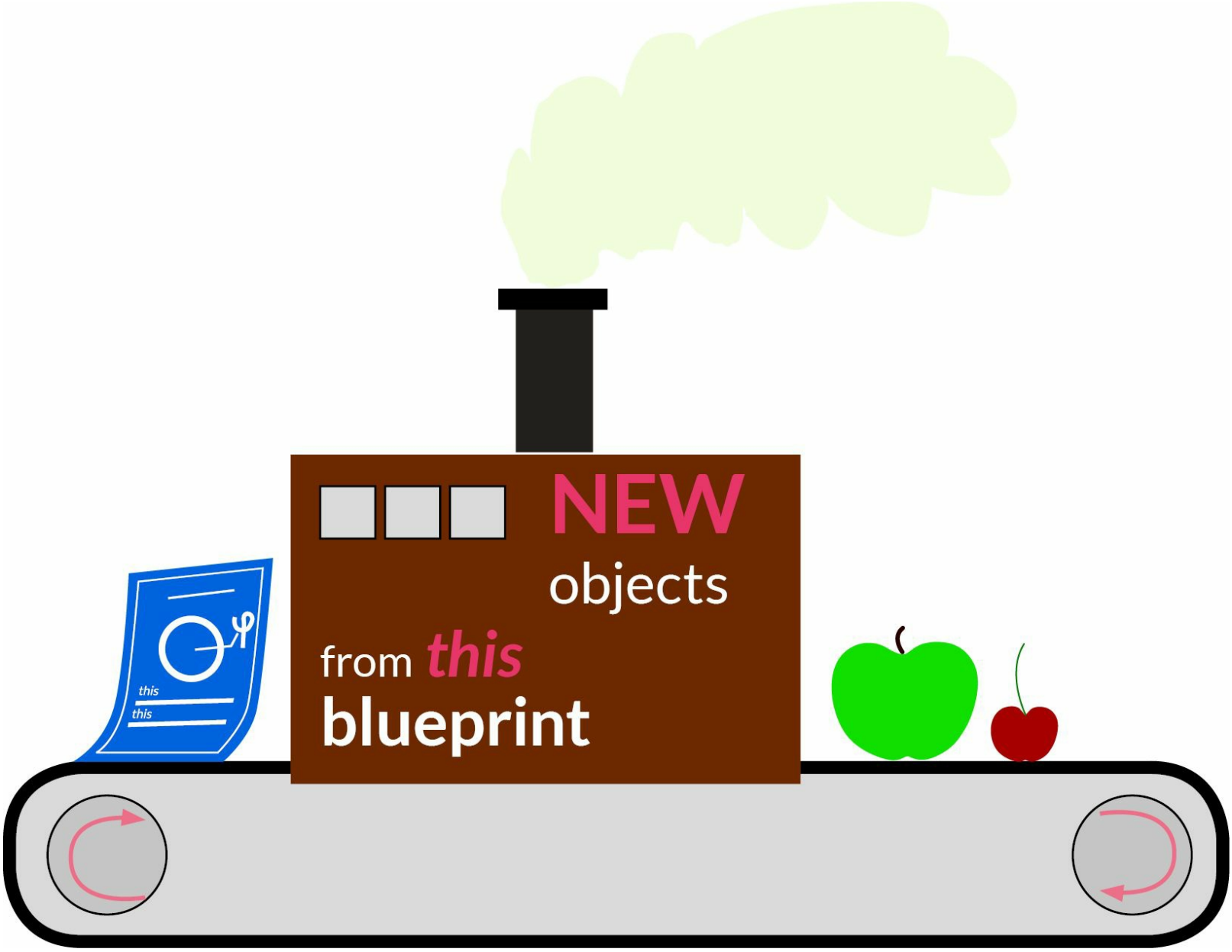
Exercise: 35
Methods.

PROBLEM: JavaScript has some predefined methods such as 'toUpperCase()' to convert lowercase strings to uppercase.

Use it to convert your name to uppercase.

ANSWER: jsvisually.com/35.html

Constructors.



Constructors are functions used to create multiple instances of an object by using the ‘new’ keyword. We have talked about them briefly on the Objects chapter. We saw a few ways to create objects, yet they were limited to one object creations. If you need to create multiple objects of the same type, then you will need to create an object constructor, and then you can create new instances of the object with different values.

The way this works is by assigning parameters to the object variables instead of actual values from the start. That would allow us to create new objects using the new keyword and pass the values as parameters, similar to how we pass them among functions.

In a way we can think of constructors as blueprints. When using the same blueprint to create a house, all the houses will be the same; the only difference will be the paint and address. Let’s take a look at a fruit object constructor:

Constructor function starts with a capital letter by convention.

```
1. var Fruit = function Fruit(name, color) {
2.     this.name = name;   //property
3.     this.color = color;
```

```
    //Method
    This.info = function() {return “I am a “ +this.color + “ “ + this.name};
}
```

Now we can easily create a large number of instances of the same object. The keyword new sets the context of “this” to the newly created object.

```
var Cherry = new Fruit();
var Apple = new Fruit();
var Mango = new Fruit();
```

Now we have created three different fruits using the same constructor. This has saved us a lot of time and code instead of creating all three from scratch.

While the fruits are different in many regards, we have used what they have in common, to link them together to one constructor, and that is the fact that they are a fruit, and they have a name and a color.

We could further extend the constructor to add weight or any other common properties if needed.

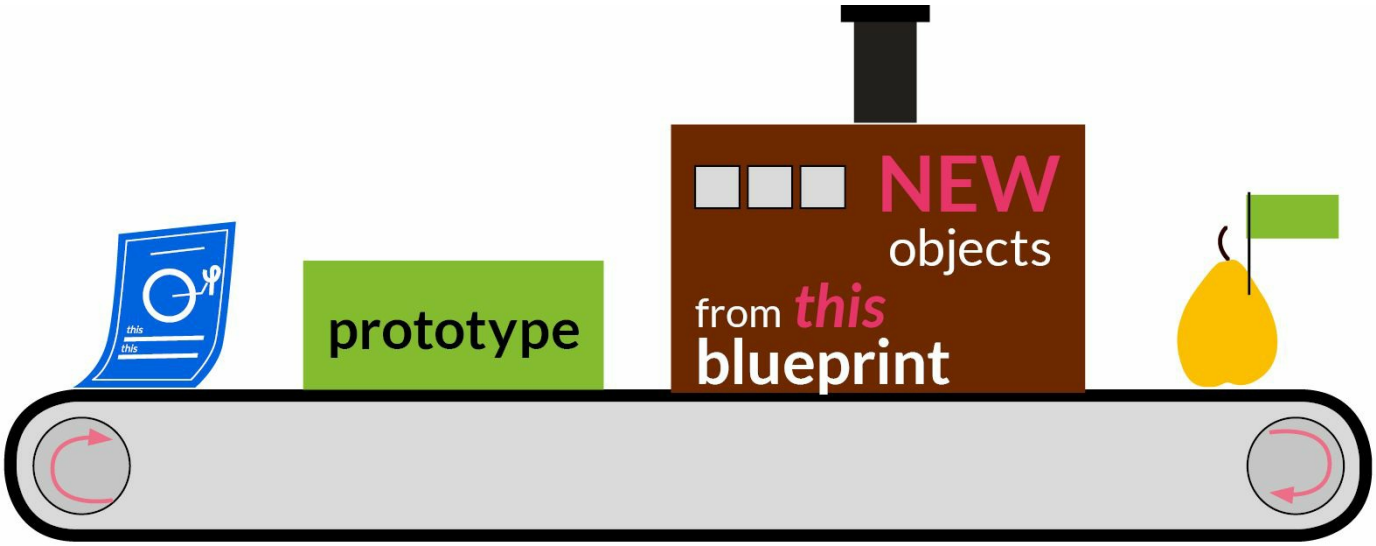
Exercise: 36
Constructors.

PROBLEM: Fix the following code:

```
1. var user = function (name) {
2.         this.name = name;
3. };
4. var john = user("John");
```

ANSWER: jsvisually.com/36.html

Prototypes.



Prototypes allow you to define properties and methods to all instances of a particular object

Every function has a property called 'Prototype'.

It allows to add properties and methods to objects, so they can share information easier.

All JavaScript objects inherit their properties and methods from their prototype. When creating objects by using an object literal or with new Object(), they inherit from a prototype called Object.prototype.

The standard way to create an object prototype is to use an object constructor function as we have done in the past. Then you would use the new keywords to create new objects from the same prototype. If you want to add new methods or properties to a prototype, then you will have to add them directly to the constructor. However, the JavaScript prototype property allows you to add new properties to an existing prototype:

```
1. var Fruit = function Fruit(name, color) {
2.     this.name = name;   //property
3.     this.color = color;
4.     //Method
5.     This.info = function() {return "I am a " +this.color + " " + this.name};
6. }
7. //This adds a property to the Fruit prototype object
8. Fruit.prototype.price = 100;
```

You can also use the prototype property to add new methods to an existing prototype:

```
1. Fruit.prototype.cost = function() {
2.     Return 'This' +this.name+'cost' + this.price;
3. };
```

You must only modify your own prototypes and not the standard ones in JavaScript.

Now let's create a new instance of the Fruit object.

```
1. var pear = new Fruit('pear', 'golden');
2. console.log(pear.price);
3. //100
4. console.log(pear.info());
5. //I am a golden pear
6. console.log(pear.cost());
7. //This pear cost 100
```

If you encounter an object and you need to get its prototype, then you can use Object.getPrototypeOf, yet, there is no such thing as Object.setPrototype.

Exercise: 37
Prototypes.

PROBLEM: The following constructor counts the amount of instances it has.

Fix the code to make it work properly

```
1. var Counter = function() {
2.         this.count++;
3. };
4. Counter.prototype.count = 0;
5. new Counter();
6. new Counter();
7. console.log((new Counter() ).count);     //should be 3
```

ANSWER: jsvisually.com/37.html