The Basics -1 (80 coding exercises)

"For the things we have to learn before we can do them, we learn by doing them." — Aristotle

1.1 What you need to know about ECMA Script

Exercises

1. What are the two ways to ensure your code is compatible in most browsers?
2. What is the difference between ECMAScript and JavaScript?

Answers

1. To ensure that your code is backward compatible with older browsers, you can transpile your code using a transpiler such as Babel. If developing an application for users of a specific device or browser it is wise to check if a particular ES feature is supported by checking online on https://www.caniuse.com.
2. ECMA Script is a general-purpose scripting language. Whereas, JavaScript is a major implementation of the ECMAScript Specification

1.2 – Variables and Data Types

Exercises:

1. When should you use **var**, **let** and **const**?
2. What is logged to the console in the following code block?

```
const game = 'Kings Quest';

game = 'Super Mario';

console.log(game);//What will be logged here?
```

3. What will the two `console.log` statements return here?

```
let num1 = 1;
function foo() {
  let num1 = 10;
  console.log(num1);
}
console.log(num1);
foo();
```

4. What is the value of `i` inside the for-loop at each iteration and outside the for-loop once iteration has ended?

```
function countI() {
  for (let i = 0; i <= 5; i++) {
    console.log(i); // 0 1 2 3 4
  }
  console.log("this is " + i); //
}
countI();
```

5. What is the value of `a` and `b` when function `scoping()` is called and why?

```
 function scoping(){
  let a = 10;
  if( a <= 10){
    var b = 5;
    a = a + 1;
  }
  console.log(a);
  console.log(b);
}
scoping();
```

6. Analyze the following block of code. What is the value of `book`?

```
{
let book = 'JavaScript is fun';
book = 'JavaScript is fun sometimes';
}
let book = 'Python is fun';
console.log(book);
```

7. Analyze the following block of code on lexical scope and guess-estimate what will happen:

```
let outer = function() {
  if (1 < 2) {
    var x = 10;
  }
  if (2 < 3) {
    var xSum = 1 + x;
    console.log(xSum);
  }

  function foo() {
    const z = 1000;
    console.log(x);
  }
  foo();
}
outer();
```

8. What is happening here with the wrapper string object?

```
let primitive = 'september';
primitive.vowels = 3;
primitive.vowels; //undefined;
```

9. What will the two `console.log` messages be?

```
console.log(typeof('universe'));

console.log(typeof(new String('universe')));
```

10. What will be logged to the console after applying the `toUpperCase()` method on the primitive string value and why?

```
let game = 'Super Mario';

game.toUpperCase();//"SUPER MARIO"

console.log(` This is a great game! ${game}`); // ?
```

11. Look at the following code. What method should you use to get the desired output?

```
let bigNum = 193934949283834949505n;

// ****** Some code here ****


// output is: "19393494928383494949505"
```

Answers

1. It is preferred to use **const** when you will not be re-assigning values. As **const** signifies a constant/unchanging value. **let**, has block-scope and is ideal for functions, loops, if conditional statements and other blocks of code. **var** can induce scoping issues and therefore should generally be avoided.

2. You cannot re-assign values to **const** variables, as this induces a `TypeError`:

```
TypeError: invalid assignment to const 'game'
```

3. The following will be logged to the console:

```
1

10
```

We first log the value of `num1` variable and in this case, `num1` has global scope. Therefore, value `1` will be logged. And lastly, we will call function `foo()`. **let** variables are block scoped, therefore inside `function foo()` , the `num1` variable has a value of `10`.

4. Inside the for-loop the value of `i` is:

```
0
1
2
3
4
5
```

Whereas, outside of the for-loop, within the function, the value `i` is:

```
ReferenceError: i is not defined
```

This is because **let** variables are block scoped and cannot be accessed outside of the for-loop block. The for-loop is demarcated by the opening and closing curly braces `{}`.

5. The following values down below will be logged for `a` and `b` respectively. This is because `a` has function scope as it has been declared with the **let** keyword. It can be accessed inside the if conditional statement. Therefore its value changes. The variable `b` is declared with **var** and can be accessed anywhere from inside the function as well.

```
11
5
```

6. The value of the variable book is `"Python is fun"`. As **let** variables can be re-assigned values and be re-declared outside of their current scope.

7. The following is logged to the console:

```
11
10
```

In order to tackle this question, simply look at the 3 blocks (function `foo()` and two `if` conditional statements).

The variable `outer` is declared with the **let** keyword and is assigned to a function. There are two if conditional statements and one function nested inside the main `outer` function assignment. Let's concentrate on the `console.log` statements.

The `console.log` statement inside the second `if` conditional statement will log the value of variable `xSum` (declared with the **var** keyword), if `1` is less than the number `2`, which indeed it is. The value of `xSum` is assigned to the sum of `1 + x`. In this case variable `x` is declared with the **var** keyword, hence it has function scope and can be accessed anywhere inside the function. The value of `xSum` is 1 added to 10, which is `11`. Moving on to function `foo()` , we are logging to the console the value of `var x` which is `10`.

8. A primitive temporary string wrapper object is created around the string value. You can add methods and properties of you own. We have added the `vowels` property to the wrapper object. However, after you apply a method or access the property of a primitive data type, the temporary wrapper object is disposed of to garbage collection and is never re-used. Which is why when trying to access the `vowels` property, `undefined` is returned.

9. The two console.log statements are:

```
"string"
"object"
```

This is because the second is an object created by the constructor function. It is uncommon to instantiate new strings with the constructor function and should be avoided.

10. The following is logged to the console:

```
" This is a great game! Super Mario"
```

The `toUpperCase()` method is applied on the wrapper object. After you apply a method or access the property of a primitive data type, the temporary wrapper object is disposed of to garbage collection and is never re-used.

11. The `toString()` method is used to display a number which is a `bigInt`, as indicated by the '`n`' at the end of the number:

```
console.log(bigNum.toString());//"193934949283834949495O5"
```

1.3 – Operators and Comparisons

1.3.1 Arithmetic operators

1.3.1.1 Operator Precedence

Exercises

1. Use the rules of operator precedence and associativity to solve the following:

```
let sum = (5 + 8 + 2) * 2;
console.log(sum); // ?
```

Answers:

1. Using associativity rules, the parenthesis `()` take precedence so `(5 + 8 + 2)` are added and the sum is then multiplied by 2. The final result is 30.

1.3.1.2 Associativity

Exercises

1. Knowing what you do about post increment operators, what will be logged to the console?

```
let cookies = 10;
console.log(cookies++); // ?
console.log(cookies); // ?
```

2. Using your knowledge of pre-decrement operators, what is the value of `cookies` in each of the `console.log` statements?

```
let cookies = 100;
console.log(--cookies);
console.log(cookies);
```

3. Try and workout what value of `i` will be logged to the console, using the pre-increment operator:

```
for(let i = 0; i <= 10; i++){
  console.log(++i)
```

```
}
```

Answers

1.

```
10
11
```

2.

```
99
99
```

3. The following is logged to the console:

```
1
3
5
7
9
11
```

The pre-increment operator will return the incremented value of `i`. During the first iteration of the for-loop, `i` is `0`. One is added to `i` and now the value is `1`. After which, `i` is incremented when the final expression of the for-loop is executed (`i++`). The for-loop will continue till `i` is 10, after which `++10` is now `11`. The final expression of the for-loop will increment `11` by `1`. The conditional statement will evaluate to false as `i` is no longer less or equal to `10`.

1.3.2 Assignment operators

Exercises

1. What will variables `num1`, `num2` and `num3` be?

```
let num1 = 100;
let num2 = 30;
let num3 = 80;
```

```
num1 = num2 = num3;

console.log(num3, num2, num1);
```

2. Solve for `x`

```
let x = 10;

let y = 20;

 x+= y*= 3;

console.log(x); //?
```

Answers

1. Following right to left associativity. This is akin to:

```
console.log(num1 = (num2 = num3)); // 80
```

```
80 80 80
```

2. `+=` and `*=` have right to left associativity. We first multiply (`y*= 3`) which is 60 and then add 10 to the value 60. Therefore, returning 70.

```
70
```

1.3.3 Comparison operators

Exercises

1. What are the two main differences between `==` and `===`?
2. Is variable `x` strictly not equal (`!==`) to `y`?

```
let x;

let y  =  10;

console.log(x !== y); //?
```

3. Compare `x` and `y` and explain why the two `console.log` messages are different?

```
let x = 0;

let y = false;

console.log(x === y);

console.log(x==y);
```

4. What is returned from the comparisons below?

```
let x = true;

let y = false;

console.log(x === y);

console.log(x == y)
```

5. What will be returned, true or false?

```
Symbol() === Symbol();
```

Answers:

1. `===` Will check if the two operands being compared are off the *same type and same value*. No type coercion is performed.

   `==` Will check if the two operands being compared are *either the same type or same value.* Type coercion is done.

2. `true`. 0 is strictly not equal to 10.

3. The following is logged to the console:

```
console.log(x === y);//false

console.log(x == y); //true
```

   In the first statement (`x === y`), the number `0` is not equal to the boolean `false`. Hence `false` is logged. Whereas, in the second statement (`x == y`). Type coercion is done and now `x` which is `0` is loosely equal to the boolean `false` value, which is represented by the number `0`. Remember, boolean `true` is represented by `1` and `false` is represented by `0`.

4. The following will be logged:

```
console.log(x === y); false

console.log(x == y); false
```

   In the first statement `x` and `y` are of different data types, hence a strict equality comparison returns `false`. The second statement will return `false` as well, since `x`, which is the boolean `true` represented by the number `1` is not loosely equal to the boolean `false` (`0`) value of `y`.

5. `false` is returned since every time a `new Symbol()` is invoked, a new unique symbol is generated.

1.3.4 Logical operators

Exercises:

1. What will the value of the variable `result` be?

```
let a = true;

let b = 0;

let result = a && b;

console.log(result);
```

2. What will be logged to the console?

```
let a  =  {
  fruit: 'lemon'
}
let b  =  {
  juice: 'mango'
}
console.log(a && b) // ?
```

3. Complete function `createUser()` that will check if the username for variable `userName` is equal to or greater than `6` characters and not an empty string. If so return `true` else, return `false`

```
let username = "Kauress";

function createUser(){

}

createUser();
```

4. What is the inverse of `!('')`

5. Is `msg1` or `msg2` logged and why?

```
const x = "null";
```

```
const y = "Bob";
function logThis() {
      if (y === "Bob" && !(x)) {
            console.log("msg1");
      } else {
            console.log("msg2")
      }
}
logThis();
```

6. What will be the value of `c` ?

```
var a = 2 ;
var b = 0;
var c = a || b;
console.log(c);
```

7. What is logged and why?

```
let user1 = {
  id: 1
}
let user2 = {
  id:2
};
let user3={
  id: 3
}
let x = 10;
if(x === 10 && typeof(x) === 'number'|| !(user1)){
console.log( user2 || user3);
```

```
} else{
  console.log('nope')
}
```

8. By looking at the following code, what do you think is returned?

```
console.log(( undefined || null || 0 ));
```

Answers:

1. `false` is logged to the console. As `true` and (`&&`) false evaluates to the boolean `false`.
2. The following will be logged to the console:

```
Object {
  juice: "mango"
}
```

3.

```
let username = "Kauress";
function createUser(){
  if(username.length >= 6 && username.trim('')){
    return(true);
  } else{
    return(false);
  }
}
createUser();
```

4. An empty string (`''`) is a falsy value. The inverse of a falsy value is `true.`
5. `msg2` is logged to the console. The inverse of `!(x)` is false. And a truthy value of non-empty string (`"Bob”)` and (`&&`) a false value will evaluate to `false`. Therefore, the condition checks if `y` which is assigned the value of `"Bob"` is equal to `false`. Since it is not, the statement in the `else` block will execute. If we

change the condition to: `(y === "Bob"&& (x))`, then `msg1` will be logged to the console.

6. `2` .Since `0` is a falsy value, the truthy `2` value will be returned

7. This question takes a bit of flexing. Don't worry. Break down the condition into small parts. At the most basic level, the condition evaluates to: `true && true || false`.
   This will then further evaluate to `true || false`, which is `true` (refer to the truthy table above). The statements in the `if` conditional statement will execute and in turn will return the first true result of `user2 || user3`. Therefore, returning `user2`

```
Object {
  id: 2
}
```

8. `0`. If all operands evaluate to falsy, then the last falsy operand is returned

1.3.5 String operator

Exercises

1. Assign a string to `let desserts` such that the output is as below:

```
"ice-cream
  cupcake
  brownies"
```

2. Fill in the function so that the string is "`Your mortagage is 2000`" is logged.

```
let payment = function() {
  function calculate() {}
  calculate();
}
payment();
```

Answers

1.

```
let desserts =
 `ice-cream
    cupcake
      brownies`;
```

2.

```
let payment = function() {
  function calculate() {
    console.log(`Your mortagage is 2000`)
  }
  calculate();
}
payment();
```

1.3.7 Typeof Operator

Exercises

1. What will `x1 === x2` result in?

```
const x1 = 2 * "abc";
const x2 = 2 * "abc";
x1 === x2; //?
```

2. What is logged to the console by both statements and why?

```
console.log(null === undefined); //?
console.log(null == undefined);   //?
```

3. What is logged to the console by both statements and why?

   Hint: This is one of JavaScript's many quirks!

```
console.log(null > 0);
```

```
console.log(null >= 0);
```

4. Explain your reasoning, why is the result is `false`

```
console.log(undefined > 0); //false
```

5. Will statements inside the `if` block execute? If so, why?

```
let x;

if(x){

  //will this return true or false

}
```

6. What does the **typeOf** operator return after the evaluation of the expression `1/10`?

```
console.log(typeof(1/10));
```

7. Does the following expression evaluate to true or false?

```
Number.isNaN(0); // T or F?
```

Answers:

1. `false`. First let's check out the value of `x1` and `x2`:

```
console.log(x1); // NaN

console.log(x2); //NaN
```

   Comparing `x1` and `x2` will return `false` as `NaN` is not equal to anything, not even itself

2. The statement `console.log(null === undefined)` will return `false` using the strict equality operator. **Null** and **undefined** are of different types and by definition one indicates the lack of a value (undefined) whereas the other indicates a value set to be non-existent on purpose (null).

   The statement `console.log(null == undefined)` will return `true` using the loose equality operator. **Null** and **undefined** when coerced return `false` and, since `false` is equal to `false`, the comparison returns `true`. According to the specification (http://www.ecma-international.org/ecma-262/5.1/#sec-11.9.3) :

   If x is null and y is undefined, return true

3. Null will be converted to the number 0, therefore, `null > 0` is `false`, since 0 is not greater than 0. Whereas `null >= 0` is `true` since 0 >= 0 is true.

4. The statement `undefined > 0` returns `false`, since **undefined** indicates a lack of a value and 0 is a numeric value.

5. `false`. This is because though `x` has been declared, it has a value of `undefined`. We will often use this in **if – else** conditional to check if an input field has some value, which is to say did it get filled in by a user.

6. The **typeof** operator will return `"number"` as the result of a mathematical calculation is a number.

7. The expression will return `false` as 0 is a number.

1.3.8 Comparisons

Exercises:

1. True or false?

```
new Number (0); // T or F?
```

2. Name the 6 values that always evaluate to false/falsy

3. True or false? Explain why?

```
 1 == "1"; //T or F?
 1 === "1"; // T or F
```

4. What will be console.logged in each case?

```
console.log(false + 1);
console.log (false == 0);
console.log(false === 0);
```

5. Change the function by adding one symbol only so that "`hello world`" is logged:

```
let a = 1;
let b = () => {
if(!(a)){
  console.log("hello world")
  }
}
b();
```

6. Filter this array of values for only truthy values:

```
const myArr = ["10", 80, true, 0, [], undefined, null,'
', NaN];
```

7. Is `NaN === NaN` true or false and why?

8. What values of `x` and `y` will be logged to the console?

```
let x = 0;

let y = 1;

if (!(x) == y) {

  console.log(x);

} else {

  console.log(y);

}
```

Answers

1. `false`. As the number `0`, corresponds to the boolean `false` value. We can check this with:

```
console.log(Boolean(0)); // false
```

2. The 6 falsey values are:
    1. '' (empty string)
    2. 0
    3. false
    4. null
    5. undefined
    6. NaN

3. The statement `1 == "1"` returns `true`, as when using the loose equality operator, the string `"1"` is coerced into the number `1`. As a result both numbers are equal to one another. Whereas, `1 === "1"` returns `false`, as both values though the same are of different data types.

4. `1`, `true` and `false`

5. The result `!(a)` , is `false`. We can use the double **NOT (!!)** operator to inverse the result of `!(a)` . Therefore, giving the actual boolean value of the operand, which is `true`

```
let a = 1;
let b = () => {
if(!!(a)){
  console.log("hello world")
  }
}
b();
```

We can also simply remove the **NOT(!)** operator, like so:

```
let a = 1;
let b = () => {
if(a){
  console.log("hello world")
  }
}
b();
```

6. We know that the last 3 values are falsy so we remove those first. And then we remove the number 0

```
let myArr = ["10", 80, true, 0, [], undefined, null,'',
  NaN];
let newArr = myArr.splice(0, 5);
console.log(newArr); // ["10", 80, true, 0, []]
```

Now let's remove the number `0` which is also a falsy value:

```
newArr.splice(3, 1);
console.log(newArr)// ["10", 80, true, []]
```

7. `false`. `NaN` is not equal to anything, not even itself.
8. The following will be logged to the console:

```
0
```

The inverse of `!(x)` is is `1`, therefore `1` is loosely equal to the value of `1` is `true` and the statements in the `if` conditional block execute.

1.4 – Coercion

1.4.2 String coercion

Exercises:

1. What does the following return?

```
console.log('41' == 41);
```

2. Evaluate the expression `12 * 6 + 'a'`. What is its data type?

3. Evaluate the following. Why do you think the answer is different from question 2?

```
let a = 'a' + 4 + 7;

console.log(a);
```

4. Explicitly coerce the boolean value `false` into a string, using the `String()` function

Answers:

1. `true`. An implicit coercion will convert the value of the number data type into a string data type. The number `41` now becomes the string `'41'`. Which returns `true` during comparison with the `==` operator.

2. `'72a'`. Multiplication takes precedence during evaluation, after which the number 72 is coerced into the string data type, and `'72'` is concatenated with `'a'`.

3. `"a47"`. All the operands have the same operator (+), therefore the expression will be evaluated moving from left to right. The number `4` will be implicitly coerced into a string and concatenated with `"a"`. Then the result `"a4"` will be concatenated with the number `7`.

4. `console.log(String(false)); //"false"`

1.4.3 Numeric coercion

Exercises:

1. What does the following return? If the return value is a string data type, explicitly coerce it into a number

```
let x = (4 * '4' + '9');
```

2. Are the results in **A** and **B** different? If so, explain why:

   A.

```
console.log(800 + ('8'));
```

   B.

```
console.log('800' + +('8'));
```

3. Re-write the following expression with numerical values instead of booleans and an empty string.

   Hint: Use the `Number()` function to coerce a value into a number.

```
let x = '';

let y = true;

let z = true;

let result = (x || y <= z)? true: false;

console.log(result) // true
```

Answers:

1. `'169'` is returned from the expression. The multiplication operator is being used, therefore the string `'4'` is converted into a number and multiplied with `4` which returns the number `16`. Moving on, the number `16` is then concatenated to the string `'9'`. This is because when using the **+** operator, instead of converting the string to a number, the number is coerced into a string instead.

   For the second part, convert the string `'169'` into a number explicitly like so:

```
Number(x); // 169
```

2. No. Notice the use of the unary + operator in B. Both return the string `'8008'`. In both A and B, the + operator signals the implicit coercion of a number to a string.

   In B the unary operator converts the string `'8'` into a number data type. This is then concatenated to the string `'800'`. Therefore, resulting in `'8008'`

3.

```
let x = 0;

let y = 1;

let z = 1;

let result = (x || y <= z)? true: false;

console.log(Number(result)); // 1
```

Alternative:

```
let result = (Number(x) || Number(y) <= Number(z))?true:false;

console.log(result);
```

1.4.4 Boolean coercion

Exercises

1. Write a function called `boolBlazer` which:
   - Takes 2 parameters `x` which is a non-boolean and `y` which is a boolean
   - Coerce the `x` parameter by using `Number()` function
   - Compares `x` and `y` using the equality (`==`) or inequality (`!==`) operator
   - If the comparison returns `true`, log the statement `'Is equal'`
   - Else if the comparison returns `false`, log the statement `'Is not equal'`
   - Pass the arguments `'hello'` and `true` to function `boolBlazer()` and call the function to execute it.

Answer

1.

```
function boolBlazer(x, y){
  if(Number(x) == y){
```

```
    console.log('Is equal')
  } else {
    console.log('Is not equal')
  }
}
boolBlazer("hello", true);
```

```
"Is not equal"
```

1.5 Iteration

1.5.1 for statement

Exercises:

1. Iterate through the numbers 1 to 10, and at each iteration print `i`

2. Iterate through the numbers 1-5, and at each iteration multiply `i` by 3

3. Can you explain what is happening in the code block below? *Notice the semicolon just before the opening curly brace.*

```
for(var i = 10; i >= 0; i--);{
  console.log(i);
for(const i = 0; i <= 10; i++){
  console.log(i) //0
}
```

4. Analyze the following code, and explain why the for-loop does not execute after the first iteration:

5. A company has a list of employees and their salary in separate arrays. You are tasked with the job of printing out the name of each employee and the employee's salary as a string in the format: `'Employee name: Employee salary:'`

```
let employees = ['Lara', 'Sukhi', 'Evee', 'Simi','Beno',
  'Jay'];
```

```
let employeeSalary = [1000, 1300, 957.89, 3230.14, 750,
  13900];
```

6. Using a nested for-loop print the following pattern:

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

7. Solve for `num` in the following block of code:

```
let num =0

for(let i = 0; i <= 2; i++){

  for(let k = 0; k <=2; k++){

    num++

  }

}

console.log(num);
```

Answers

1.

```
for(let i = 0; i <= 10; i++){

  console.log(i);

}
```

2.

```
for(let i = 0; i <= 5; i++){

  console.log(i * 3);

}
```

3. The for-loop has three *expressions*, which are separated by semicolons. Since the semicolons only separate the expressions from each other, only two are required.

The extra third semi-colon is interpreted as a separator for a fourth expression. This is improper syntax.

4. Variables declared with **const** cannot be re-assigned values, and an error will be outputted:

```
TypeError: invalid assignment to const 'i'
```

5. The for-loop will iterate over the indices of the first array, and use that to index into the others.

```
let employees = ['Lara', 'Sukhi', 'Evee', 'Simi','Beno',
   'Jay'];

let employeeSalary = [1000, 1300, 957.89, 3230.14, 750,
   13900];

for(let i= 0; i < employees.length; i++){

  console.log(employees[i] + employeeSalary[i])

}
```

This will log the following to the console:

```
"Lara1000"

"Sukhi1300"

"Evee957.89"

"Simi3230.14"

"Beno750"

"Jay13900"
```

6.

```
for(let i=0; i<=5; i++){

  for(let j = 0; j <= i; j++){

    document.write("*");

  }

      document.write('<br>');

}
```

7. The value of `num` is 9. The inner loop will iterate 3 times (it completes 1 cycle) for each time that the outer loop iterates. At each iteration within the inner loop the value of `num` is incremented by 1, for a total of 3 times

```
console.log(num);// 9
```

1.5.2 While statement

Exercises

1. Is the following statement true or false?:

   "A while loop will execute for as till a conditional statement becomes false. "

2. Write the conditional clause in this while loop that will cause the browser window to crash

```
let x;
while(){
  console.log("Infinite Loop")
}
```

3. Write a function called `magicBall()` that will display 3 random numbers to a user. Use a `while` loop to complete the code.

```
function magicNumber(){
}
magicNumber();
```

Answers:

1. True

2.

```
let x;
while(true){
  console.log("Infinite Loop");
}
```

3.

```
function magicBall(){
let chance = 1;
while(chance <=3){
let magicNumber= Math.floor(Math.random() * 11);
chance++
console.log(magicNumber);
 }
}
magicBall();
```

1.5.3 Do while statement

Exercises

1. Code a **do-while** loop which will print the number 0, five times.
2. Write a **do-while** loop that will execute once and log a string `'Love and Peace'` with the conditional `x = false`. Do not change the value of `x` inside the while statement.
3. Modify the function in question 2, to infinitely execute a **do-while** loop. You may change the value of `x`.
4. Write a function that takes `x` as a parameter. `x` refers to the number of tickets sold at a local cinema. Within the body of the function use a **do-while** loop to inform the ticket attendant know how many seats and tickets are available, given a fixed number of seats (30).

```
function ticketSold(x){
  let availableSeats = 30;
  let tickets = x;
  // your code here
}
```

```
ticketSold(40);
```

Answers

1.

```
let zero = 0;
let count = 0;
do{
  console.log(zero);
  count++
} while(count < 5);
```

2.

```
let x = false;
do{
  console.log('Love and Peace');
  x = true;
} while (x = false)
```

The **do-while** loop will execute at least once, before the conditional statement is evaluated.

3.

```
let x = true;
do{
  console.log('Love and peace');
}while(x );
```

4.

```
function ticketSold(x){
let availableSeats = 30;
let tickets = x;
```

```
  do{
    availableSeats--;
    tickets--
    console.log('available seats:' + availableSeats + "
tickets left:" + tickets);

  }while(availableSeats >= 1 && tickets >= 1);
}
ticketSold(40);
```

1.5.4 Break and continue statements

Exercises

1. What is the key difference between the **break** and **continue** statements?
2. Print out all odd numbers from `0-10` using the **break** statement

3. Iterate over each letter of the 3 letter words in the following array. Logging all the letters in a word on a new line, except for vowels (“a”, “e”, “i”, “o”, “u”). Use a **continue** statement.

```
let wordList=['umbrella', 'apple', 'paint', 'never', 'outmost'];
```

Answers

1. The **break** statement is used to exit a loop completely, once a condition has been met, or after *x* number of iterations. Whereas, the **continue** statement will check a condition and terminate the current instance of an iteration in a loop if the condition is true. And then continue iterating.
2.

```
for(let i = 0; i <=10; i++){
  if(i % 2 === 0){
   continue;
  }
```

```
   console.log(i);
}
```

3.

```
let wordList = [
'umbrella',
'apple',
'paint',
'never',
'outmost'
];
for (let i = 0; i < wordList.length; i++) {
  let currentname = (wordList[i]);
  for (let j = 0; j < currentname.length; j++) {
    if (currentname[j] === 'a' ||
      currentname[j] === 'e' ||
      currentname[j] === 'i' ||
      currentname[j] === '' ||
      currentname[j] === 'o' |
      currentname[j] === 'u') {
      continue
    }
    console.log(currentname[j]);
  }
}
```

1.6 – Conditional statements

Exercises

1. Write the syntax of an `if - else` block
2. Write a conditional statement that will check if:
    1. The undefined and null data types are strictly equal (`===`) to each other
    2. Undefined and null are loosely equal (`==`) to each other
3. Code a function that will accept a user's age as input. The function must log different messages depending on the user's age. Use the **if**, **else if**, **else** statements.

    Listed below are the conditions:

    Ages: `0 - 18:` Message: "Please get parental permission"

    Ages: `19 - 30:` Message: "You will get an email within seven days"

    Ages `31 - 40:` Message: "Please check your email in 3 days"

    Ages `41 - 50:` Message: "Please login in a few hours"

    Ages `51 - 60` Message: "You are now enrolled"

    Ages `60+` Message: "Please continue to login"

    Note: declare a variable called userage and evaluate it against different conditions
4. Refactor the above question (#3) using a switch statement with various cases
5. What is returned from the following code statement?

```
console.log("0" == false? true: false);
```

6. Refactor the following block of code which will check if `x` is smaller than variables `y` and `z`. Change the function in the following ways:
    1. Declare the variables in global scope
    2. Use a ternary operator

    Note: You need not use a function

```
let kawaii = function(){
  let x = 10;
```

```
  let y = 20;
  let z = 30;
  if (x < y){
    console.log('x is smaller than y');
  }
   if (x < z){
    console.log('x is smaller than z');
  }  else  {
    Return false;
  }
}
kawaii();
```

Answers:

1.

```
if (condition) {
   } else {
   }
```

2.

```
if(undefined === null) {
  console.log('Strictly not equal');
} else {
  console.log('Loosely equal');
}
console.log('Loosely equal')
}
```

3.

```
let userage = 61;
if(userage > 0 && userage <= 18) {
  console.log('Please get parental permission')
} else if (userage > 18 && userage <= 30){
  console.log('You will get an email within seven days')
} else if (userage > 30 && userage <= 40) {
    console.log('Please check your email in 3 days')
} else if (userage > 40 && userage <= 50) {
    console.log('Please login in a few hours')

} else if (userage > 50 && userage <= 60) {
    console.log('You are now enrolled')

} else {
  console.log("Please continue to login")
}
```

4.

```
  let userage = 30;
  switch (true) {
   case userage > 0 && userage <= 18:
    console.log('Please get parental permission');
    break;
   case userage > 18 && userage <= 30:
    console.log('You will get an email within seven days');
    break;
```

```
 case userage > 30 && userage <= 40:
  console.log('Please check your email in 3 days');
  break;
 case userage > 40 && userage <= 50:
  console.log('Please login in a few hours');
  break;
  case userage > 50 && userage <= 60:
  console.log('Please login in a few hours');
  break;
 default:
  console.log('Please continue to login');
}
```

5. `true`. The boolean equivalent of string `"0"` is `false`.

6.

```
let x = 100
let y = 20;
let z = 30;
x < y ? console.log('x is smaller than y') : false;
x < z ? console.log('x is smaller than z') : false;
```