2. Using comment blocks:

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

Week 7

Objectives

1. Conditionals
2. Loops
3. Functions
4. Comparing values
    a. Coercion
    b. Truthy and Falsy
    c. Equality
    d. Inequality

7.1 Conditionals

Making decisions in your code — conditionals - Learn web development | MDN (mozilla.org)

Conditional statements are used to perform different actions based on different conditions, think of it as how code makes decisions. Different conditions can be evaluated using **boolean expressions**. Boolean expressions can be created using **relational operators**. For example:

```
5 > 6       //false
7 < 210     //true
9 >= (36/4) //true
```

Conditional statements allow us to represent decisions that we make in our daily lives in Javascript. Javascript allows us to express conditional statements in the following ways:

1. If…else statements
2. Switch statements, and
3. Ternary operator

7.1.1 If…else statements

Basic syntax of an if statement:

```
if (condition) {
  /* code to run if condition is true */
} else {
  /* run some other code instead */
}
```

The else part is not required, hence the following syntax is also valid

```
if (condition) {
  /* code to run if condition is true */
}

/* run some other code instead */
```

Here, the code after the if statement is run whether or not the condition passes(is true) or not(false). The third discouraged way of writing an if statement is without the curly braces.

```
if (condition) /* code to run if condition is true */
else /* run some other code instead */
```

It is perfectly okay to nest if…else statements. Such a scenario may have the following syntax normally to test for different conditions that should work together in some way

```
if (condition) {
  /* code to run if condition is true */
  if (condition) {
    /* Nested if statement */
  }
} else {
  /* run some other code instead */
}
```

Compound conditions are allowed in if statements. We create compound conditions by combining two or more boolean expressions using logical operators. The syntax for an if statement making use of a compound condition may look something like this

```
if( condition && condition ){
  /* code to run if compound condition is true */
}
```

7.1.2 Switch statements

If…else statements get the job done but can get cumbersome when you want to set a variable to a particular value or you want to print out a particular statement depending on the statement. Switch statements take in a single expression/value and look through a sequence

of choices executing one that matches the value executing it and whatever code comes along with it. The syntax of an switch statement is as follows:

```
switch (expression) {
  case choice1:
    run this code
    break;

  case choice2:
    run this code instead
    break;

  // include as many cases as you like

  default:
    actually, just run this code
}
```

7.1.3 Ternary Operator

It is a syntax that tests out a condition and spits out an expression/value depending on whether the condition is true or false.

```
( condition ) ? /*run this code*/ : /*run this code instead*/
```

7.2 Loops

[Loops and iteration - JavaScript | MDN (mozilla.org)]

Loops offer a quick and easy way to do something repeatedly. Iteration statements are used to represent loops/iterations.

There are many kinds of loops, characterized by how they determine the start and end points of the loop. There are situations that are served easily by one type of loop over the other.

The type of loop statements provided by Javascript are:

1. for statement
2. do...while statement
3. while statement
4. labeled statement
5. break statement
6. continue statement
7. for...in statement
8. for...of statement

7.2.1 for statement

For loop executes until the condition returns false. It has the following syntax:

```
for ([initialExpression]; [conditionExpression];
[incrementExpression])
  statement
```

7.2.2 do…while statement

It has the following syntax:

```
do
  statement
while (condition);
```

The statement is executed at least once before the condition is checked

7.2.3 while statement

Syntax:

```
while (condition)
  statement
```

7.2.4 labeled statement

Provides a statement with an identifier that lets you refer to it elsewhere in your program. For example, you can use a label to identify a loop and then use `break` or `continue` to indicate whether a program should interrupt a loop or continue its execution.
The syntax of a labeled statement is as follows:

```
label:
  statement
```

The value of statement may be any Javascript identifier that's not a reserved keyword.

For example:

```
markLoop:
while(theMark == true){
  doSomething();
}
```

7.2.5 `break` statement

Use the break statement to terminate a loop or in conjunction with a labeled statement.

- When used without a label, it terminates the innermost enclosing loop or switch immediately and transfer control to the statement that follows
- When break is used with a label, it terminates the specified labeled statement.

Syntax:

```
break;
break [label];
```

7.2.6 continue statement

Can be used to restart a loop or label statement.

- When used without a label, it terminates the current iteration of the innermost enclosing loop statement and continues execution of the loop with the next iteration. In contrast to the break statement, continue jumps to the next iteration
- When used with a label, it applies to the looping statement identified with the label.

Syntax:

```
continue;
continue [label];
```

7.2.7 for…in statement

Iterates a specified variable over all the enumerable properties of an object. For each property, Javascript executes the specified statements.

Syntax:

```
for(variable in object)
      Statement
```

A caveat with `for…in` when working with Arrays is that it iterates over user-defined properties as well as the array elements. This is an edge case that happens if you modify the Array object, for example by adding custom properties or methods.

7.2.8 for…of statement

Creates a loop iterating over iterable objects (e.g. Array, Map, Set, arguments object), invoking a custom iteration hook with a statement to be executed for the value of each distinct property.

```
for (variable of object)
      statement
```

While a `for…of` loop iterates over property values, a `for…in` loop iterates over property names.

7.3 Functions

Functions - JavaScript | MDN (mozilla.org)

Functions are one of the fundamental building blocks in JavaScript. It is a set of statements that perform a task. To use a function, it must be defined somewhere in a scope that you wish to call it.

7.3.1 Defining functions

7.3.1.1 Function declarations

A function definition consists of a function keyword followed by:

1. Name of the function
2. List of parameters of the function, enclosed in parentheses and separated by commas.
3. Javascript statements that define the function, enclosed in curly brackets.

Syntax

```
function square(number) {
  return number * number;
}
```

Parameters are passed to function by value - if the code within the body of a function assigns a new value to a parameter the change is not reflected globally or in the code that called the function. If an object is passed as a parameter, and the function statements change the object properties, the change is visible outside the function. This also happens with arrays where if a function receives an array as a parameter and the function statements change any of the array's value, the change is visible outside the function.

7.3.1.2 Function expression

A function created by a function expression can be anonymous (it does not have a name). For example

```
const square = function(number) {return number * number}
var x = square(4)
```

However, a name can be provided with a function expression. Providing a name allows the function to refer to itself and also makes it easier to identify the function in a debugger stack trace.

```
const factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1) }
console.log(factorial(3))
```

Function expressions are convenient when passing a function as an argument to another function.

Functions can be defined at run time from strings using the `Function` constructor.

7.3.2 Calling functions

Defining functions does not execute them. It names the function and specifies what to do when the function is called.

Calling the function performs the specified actions with the indicated parameters. For example:

```
square(5);
```

Functions must be in scope when they are called. The scope of a function is the function in which it is declared or the entire program if declared at the top level. **Function declarations** can be hoisted (appear below the call in the code). Note that function hoisting only works with function declarations and not with function expressions.

A function can call itself (recursion).

7.3.3 Function scope

Variables inside a function cannot be accessed from anywhere outside the function. This is because, the variable is defined only on the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined.

7.3.4 Closures

Javascript allows for nesting of functions and grants the inner function full access to all the variables and functions defined inside the outer function. This includes all other variables and functions that the outer function has access to.

However, the outer function does not have access to the variables and functions defined in the inner function. Since the inner function has access to the scope of the outer function, the variables and functions defined in the outer function will live longer than the duration of the outer function execution, if the inner function survives beyond the life of the outer function.

A closure is created when the inner function is made available to any scope outside the outer function.

7.3.5 Using arguments object

The arguments of a function are maintained in an array-like object. Within a function you can address the arguments passed to it as follows

```
arguments[i]
```

Where i is the ordinal number of the argument, starting at 0. The first argument passed to a function would be `arguments[0]` and the total number of arguments is indicated by `arguments.length`. This is often useful if you do not know in advance how many arguments will be passed to the function. You can use `arguments.length` to determine the number of arguments passed to the function, and then access each argument using the `arguments` object.

For instance, consider a function that concatenates several strings.

7.3.6 Function parameters

7.3.6.1 Default parameters

Parameters of functions default to undefined. However in some situations it might be useful to set a different default value.

Without default parameters
The general strategy for setting default was to test parameter values in the body of a function and assign a value if they are `undefined`. Consider the following example

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;

  return a * b;
}

multiply(5); // 5
```

With default parameters
A manual check in the function body is no longer required. Consider the following example

```
function multiply(a, b = 1) {
  return a * b;
}

multiply(5); // 5
```

7.3.6.2 Rest parameters

The rest parameter syntax allows us to represent an indefinite number of arguments as an array. Consider the following example

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map(x => multiplier * x);
}

var arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

In the example, the function uses rest parameters to collect arguments from the second one to the end. The function then multiplies these by the first argument.

7.3.5 Arrow functions

An arrow function expression has a shorter syntax compared to function expressions and does not have its own this, arguments, super or new.target. Arrow functions are always anonymous.

Arrow functions are important for the following reasons:

1. Shorter functions
2. Non-binding of `this`

7.3.5.1 Shorter functions

In some functional patterns shorter functions are welcome. Compare:

```
var a = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

var a2 = a.map(function(s) { return s.length; });

console.log(a2); // logs [8, 6, 7, 9]

var a3 = a.map(s => s.length);

console.log(a3); // logs [8, 6, 7, 9]
```

Further reading:

1. Javascript's predefined functions
2. Scope and the function stack

7.4 Comparing values

7.4.1 Coercion

Type coercion is the automatic or implicit conversion of values from on data type to another. For instance converting strings to numbers. Type conversion is similar to type coercion because they both convert values from one datatype to another with one key difference - type coercion is implicit whereas type conversion can either be implicit or explicit.

7.4.2 Truthy and Falsy

In the context of Javascript, a truthy value is one that can be considered true when encountered in a boolean context. All values are truthy unless they are defined as falsy. That is all values are truthy except `false`, `0`, `-0`, `0n`, `“”`, `null`, `undefined` and `NaN`.

7.4.3 Equality

The equality operator (==) checks whether its two operands are equal, returning a Boolean result. Unlike the strict equality operator, it attempts to convert and compare operands that are of different types.

7.4.4 Inequality

The inequality operator (!=) checks whether its two operands are not equal, returning a Boolean result. Unlike the strict inequality operator, it attempts to convert and compare operands that are of different types.