Function Sequence

JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.

```
<!DOCTYPE html>
<html>

<body>

  <h1>JavaScript Functions</h1>
  <h2>Function Sequence</h2>
  <p>JavaScript functions are executed in the sequence they are
called.</p>

  <p id="demo"></p>

  <script>
   function myDisplayer(some) {
     document.getElementById("demo").innerHTML = some;
   }

   function myFirst() {
     myDisplayer("Hello");
   }

   function mySecond() {
     myDisplayer("Goodbye");
   }

   myFirst();
   mySecond();
  </script>

</body>

</html>
```

Sequence Control

Sometimes **you would like to have better control over when to execute a function.**

Suppose **you want to do a calculation**, and **then display the result.**

You could call a calculator function (myCalculator), save the result, and then call another function (myDisplayer) to display the result

```
<!DOCTYPE html>
<html>

<body>

  <h1>JavaScript Functions</h1>
  <h2>Function Sequence</h2>
  <p>JavaScript functions are executed in the sequence they are called.</p>

  <p>The result of the calculation is:</p>
  <p id="demo"></p>

  <script>
   function myDisplayer(some) {
     document.getElementById("demo").innerHTML = some;
   }

   function myCalculator(num1, num2) {
     let sum = num1 + num2;
     return sum;
   }
   let result = myCalculator(5, 5);
   myDisplayer(result);
  </script>

</body>

</html>
```

```


<script>
 function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
 }

 function myCalculator(num1, num2) {
  let sum = num1 + num2;
 myDisplayer(sum);
 }

 myCalculator(5, 5);
</script>
```

The **problem** with the **first example** above, is that **you have to call two functions to display the result**.

The **problem** with the **second example,** is that **you cannot prevent the calculator function from displaying the result.**

Now it is time to bring in a callback.

JavaScript Callbacks

A callback is a function passed as an argument to another function.

```
<script>
  function myDisplayer(something) {
    document.getElementById("demo").innerHTML =
something;
  }

  function myCalculator(num1, num2, myCallback) {
    let sum = num1 + num2;
    myCallback(sum);
  }

  myCalculator(5, 5, myDisplayer);
 </script>
```

In the example above, myDisplayer is a called
a **callback function**.

It is passed to myCalculator() as an **argument**.

```
function myDisplayer(something) {
  document.getElementById("demo").innerHTML = something;
}

function myCalculator(num1, num2, myDisplayer) {
  let sum = num1 + num2;
  myDisplayer(sum);
}

myCalculator(5, 5, myDisplayer);
```

```
<script>
 function myDisplayer(something) {
  document.getElementById("demo").innerHTML =
something;
}

 function myCalculator(num1, num2, myFunction) {
  let sum = num1 + num2;
 myFunction(sum);
}

 myCalculator(5, 5, myDisplayer);
</script>
```

Note

When you pass a function as an argument, remember **not to use parenthesis.**

JavaScript Promises

Promise Object Properties

A JavaScript Promise object can be:

- **Pending**
- **Fulfilled**
- **Rejected**

The Promise object supports two properties
: **state** and **result**.

- While a Promise object is **"pending"** (working), the **result is undefined.**
- When a Promise object is **"fulfilled",** the result is a **value.**
- When a Promise object is **"rejected",** the result is an **error object.**

```
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Promise.then() **takes two arguments**, a callback for success and another for failure.

Async Syntax

The keyword async before a function **makes the function return a promise**

Await Syntax

The await keyword can only be used inside an async function.

The await keyword **makes the function pause the execution and wait for a resolved promise**

```
async function myDisplay() {
  let myPromise = new Promise(function (resolve, reject) {
    resolve("I love You !!");
  });
  document.getElementById("demo").innerHTML = await
myPromise;
}

myDisplay();
```

Callback Hell

The phenomenon which happens when we nest multiple callbacks within a function is called a callback hell.

Callback Hell

Callback hell is a phenomenon **where a Callback is called inside another Callback.** It is the nesting of multiple Callbacks inside a function. If you look at the design of the code, it seems just like a pyramid. Thus, the Callback hell is also referred to as the **'Pyramid of Doom'**.