50.003 - Basics of Web Programming

Learning Outcomes

By the end of this unit, you should be able to

1. Define an HTML document with elements and attributes.
2. Define the cosmetic and layouts of an HTML documents using CSS.
3. Use JavaScript to capture user inputs from HTML documents.
4. Use JavaScript to render output to the HTML documents.
5. Apply JavaScript operators to compute results
6. Define customized JavaScript datatype using class
7. Define named and anonymous functions in JavaScript

Basic Components of A Web Application

A Web application consists of two major parts.

1. Front-end - the user interface, is often presented with a browser on the PC or on the mobile devices. The front-end loads and presents the required data to the users. The front-end also captures the user's inputs and translates them into system actions to be performed by the web application (both frontend or backend).
2. Back-end - the application's brain. It is often made up of the business logic layer and the data layer. The data layer takes care of the data storage, query and manipulation, while the business logic layer implements the defined actions to be performed given the user inputs (from the front-end).

Common front-end software components

1. HTML
2. CSS
3. JavaScript

Common back-end software components

1. A database or a set of databases
2. A backend application server or a set of back-end application servers
3. A set of cache storage (optional)
4. A group of load balancers (optional)

HTML

Hyper Text Markup Language is a language to define the structure of a document (AKA a web page).

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>50.003</title>
  </head>
  <body>
    <p class="heading">Welcome to 50.003!</p>
    <p id="msg1" class="message">It's about Software Engineering!</p>
  </body>
</html>
```

There are many versions/standards of HTML. We only consider HTML5 in this unit. An HTML document consists of a doctype declaration, which tells the browser which version of HTML it is in. The second part is an `<html>` element.

Each element consists of an open tag and a close tag and the children. For instance, `<html>` is the opening tag and `</html>` is the closing. The children of an elemnent can be elements or texts. However there are a set of rules governing what contents could appear in a given element. In the above example, the content of the `<html>` element consists of a `<head>` element and a `<body>` element. Element without child/content can be written in a short form, e.g. `<meta charset="utf-8" />` is a short hand of `<meta charset="utf-8"></meta>`. Note that `charset="utf-8"` denotes an attribute of this `<meta>` element.

The HTML document in the above example defines a structure as follows,

- html
- head
  - meta
  - title
  - text
- body
  - paragraph
  - text

CSS

Cascading Style Sheets is a language used in describing the presentation of a document. Being different from HTML, it focuses on the aesthetical requirement of the documents. For instance, it defines the answers to the following questions,

1. Which font family should the title text be rendered in?
2. How many pixels should the text margin be?
3. ... and etc.

```
p {
  margin: 10px;
}

#msg1 {
  font-family: "Courier New", Courier, monospace;
}

.heading {
  font-size: larger;
}
```

There are three rules in the above CSS. Each rule consists of two parts.

1. terms preceding the `{` are called the selectors.
2. terms being enclosed by `{` and `}` are called the declarations.
3. Each declaration consists of a property (preceding the colon `:`) and a value (following the colon `:`).
4. Declarations are terminated by a semi colon `;`.

For instance, in the first rule, we find `p` as the selector, which means that this rule is applicable to all `<p>` elements in the target HTML document. For all applicable elements, we want to set the margin to be 10 pixel wide. The second rule applies to a particular element with id = `#msg1`. Note that ids in an HTML document must be unique. In this rule, we want to set the font family of the text contained in the applicable elements to `Courier New`. In the third rule, we would like to apply the declaration to all elements with `class` attribute equal to `heading`.

Suppose the above css definitions are stored in a file named `hello.css`, we could add the following as one of the child element of the `<head>` element in the earlier html document.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>50.003</title>
    <link rel="stylesheet" href="hello.css" />
  </head>
  <body>
    <p class="heading">Welcome to 50.003!</p>
    <p id="msg1" class="message">It's about Software Engineering!</p>
  </body>
</html>
```

Note that selectors are *composable*, e.g. selector `p.heading` selects all `<p>` elements with `class` equal to `heading`.

For information refers to

```
https://www.w3schools.com/cssref/css_selectors.php
```

Overlapping CSS rules

Suppose we add a new CSS rule to the earlier CSS

```
p {
  margin: 10px;
}

#msg1 {
  font-family: "Courier New", Courier, monospace;
}

.heading {
  font-size: larger;
}
// new rule
p.message {
  margin: 5px;
}
```

The first and the newly added rules are overlapping, i.e. the set of HTML elements they select are overlapping, i.e.

```
<p id="msg1" class="message">It's about Software Engineering!</p>
```

is selected by both rules. However the declarations of the two rules are not compatible. In such a situation, CSS interpretors will pick the one with higher specificity to apply the selected HTML element.

For details on how specificity score is calculated, refer to the following

```
https://www.w3.org/TR/CSS21/cascade.html#specificity
```

JavaScript

JavaScript is a programming language that was initially designed to run in the browsers. Its internal design was influenced by Lisp and Scheme, it s external syntax was strongly influence by Java, though its behavior differs from Java. In the last decade, JavaScript was extended to run directly in the backend systems as well. In this unit, we consider the version of JavaScript that run in the browsers.

A Simple Example

First we modify the HTML document introduced earlier to include a `<script>` element. Note that we can't use `<script/>` short-hand as it won't work in certain browsers.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>50.003</title>
    <link rel="stylesheet" href="hello.css" />
    <script src="hello.js"></script>
  </head>
  <body>
    <p class="heading">Welcome to 50.003!</p>
    <p id="msg1" class="message">It's about Software Engineering!</p>
  </body>
</html>
```

The content of the `hello.js` file is as follows.

```
console.log("hello");
```

In the above, we define a JavaScript program that prints a string `hello` in the console. Note that the printed string will not appear in the HTML page, instead it is printed in the console of the browser

- Chrome and Edge: "three vertical dot" -> "More Tools" -> "Developer Tools" -> "Console"
- Firefox: "three horizontal bars" -> "More Tools" -> "Web Developer Tools" -> "Console"
- Alternatively, we may also use online learning tools such as

```
https://jsfiddle.net/
```

To display "hello" message in a pop-up window, we could change the `console.log()` method with `alert()`.

```
var msg = "hello";
// console.log(msg);
alert(msg);
```

In the above, we have 3 lines code JS codes. In the first statement we define a variable `msg` with value `"hello"`. In JavaScript `var` keyword in define a variable. The second statement the `console.log()` that we used previously and it is commented away. The third statement is a method call to `alert()` with `msg` as the actual argument.

Lastly, if we want to add a string to the begining of the HTML document, we may use `document.write()` method.

```
var msg = "hello";
/* console.log(msg);
alert(msg); */
document.write(msg);
```

In the above we use `/* ... */` to comment multiple lines. `document` is a predefined variable holding the reference of the current HTML document. `document.write()` takes a string and inserts it to the begining of the document.

Note that the semi-colon at the end of each statment is optional.

Input/Output

One may ask, "what about displaying the message inside some of the HTML element?"

First attempt

Suppose we change the Javascript program as follows

```
var msg = "hello";
var p_elem = document.getElementById("msg1");
p_elem.innerText = msg;
```

In the second statement, we call `document.getElementById()` to retrieve the reference to the element with id= `msg1` in the HTML document and store the reference in a variable `p_elem`.
In the third statement, we set the inner HTML of this element as `msg`.

Howe when we load the HTML with the above script, some error occured.

```
Uncaught TypeError: p_elem is null.
```

This is because we retrieve the element reference before the document is fully rendered hence `p_elem` is having a null reference.

Second attempt

To fix this problem, we need to postpone the JavaScript execution until =the doucment is fully rendered. There are two possible ways to do that.

1. Write a busy loop to keep checking whether the document is fully rendered.
2. Enclose the action (of updating the content of the `<p id="msg1">` element) in a function. Make this function into a call-back, which will only be invoked when the document is done with rendered.

To keep the resource usage low, JavaScript adopts the 2nd idea.

```
var msg = "hello";
function run() {
  var p_elem = document.getElementById("msg1");
  p_elem.innerText = msg;
}
document.addEventListener("DOMContentLoaded", run);
```

In the above, we define a function named `run` with no argument. The body of the function contains the action to be performed when the document is fully rendered. In the last statement, we *register* the `run()` function as the call-back of the event "DOMContentLoaded" of the `document` object. The function `run()` is able to access the variable `msg` defined outside. In JavaScript this is called *closure*.

Capturing the user input

Now let's add a textbox in the HTML document and capture the user's input in this textbox via a button click. Consider `textandbutton.html` as follows,

```
<!DOCTYPE html>
<html>
  <head>
    <script src="textandbutton.js"></script>
    <meta charset="utf-8" />
    <title>50.003 sample code: Text and Button</title>
  </head>
  <body>
    <div>Your input: <input id="textbox1" type="text" /></div>
    <div>Output: <span id="span1"></span></div>
    <div><button id="button1">Submit</button></div>
  </body>
</html>
```

In the above HTML document, we find three components in the body, an `<input>`, a `<span>` as output field and a `<button>`.

In the `textandbutton.js` we have the following definitions.

```
function handleButton1Click() {
  var textbox1 = document.getElementById("textbox1");
  var span1 = document.getElementById("span1");
  span1.innerHTML = textbox1.value;
}

function run() {
  var button1 = document.getElementById("button1");
  button1.addEventListener("click", handleButton1Click);
}

document.addEventListener("DOMContentLoaded", run);
```

JavaScript Variable

There are 2 ways to declare a variable: `var` and `let`

Use `const` to declare a constant

`let` and `const` was introduced in ES6 (ECMAScript 6) in 2015. ECMAScript is a language standard for JavaScript.

The main difference between `var` and `let` mainly is the scope. `var` can be a global scope or function scope, while `let` is a block scope, meaning it only exists inside a code block. A code block is defined by curly brackets { }. Also, `var` variable can be redeclared, overriding the existing variable.

When should you use `var` or `let`? Most of the time it does not matter. However, using `let` is preferred to prevent accidental variable overriding and it has a better memory management. Use `var` if the variable scope needs to be global or in the whole function. The code below demonstrate the difference:

```
var varA = "varA"; // Global Scope
let letA = "letA"; // Global Scope

function globalAccess() {
  console.log(varA); // Accessible anywhere
  console.log(letA); // Accessible anywhere
}
globalAccess();

function outerFunc() {
  if (1 < 2) {
    // Curly bracket defines a block
    var varB = "varB"; // Function Scope
    let letB = "letB"; // Block Scope
  }
  console.log(varB); // varB is accessible
  console.log(letB); // Error, letB is NOT accessible
}
outerFunc();
```

Function and Anonymous Function

The syntax of function is as follows:

```
function functionName(parameters) {
  // function body
  return output;
}

// Invoke function
functionName(args);
```

In JavaScript, you can declare anonymous functions using either of these syntaxes:

```
function (params) {
  //function body
  return output;
}
```

or,

```
(params) => {
  //function body
  return output;
};
```

Anonymous functions are commonly use for one-time use function. You will see and use a lot of this kind of function in JavaScript.

Basic Data Structures

String

String in Javascript is a builtin datatype.

```
var myStr = "hello";
```

We can access the characters in the string `myStr` by their positions.

```
myStr[0];
```

returns a string containing the first character of `myStr`.

```
myStr[myStr.length - 1];
```

returns a string containing the last character of `myStr`. `myStr.length` computes the length of `myStr`.

```
myStr + " 123";
```

returns a new string by concatenating `myStr` to another string `" 123"`. Note that `myStr` remains unchanged. Alternatively, we could use `myStr.concat(" 123")`, which produces the same result. If we would to store the result of concatention, we need to assign it to a new variable.

```
var newStr = myStr + " 123";
```

We can also use `${}` to embed a string variable into a template string. A template string is enclosed with (`) instead of (").

```
var newStr = `${myStr} 123`;
```

We may split a string by `.split()` method.

```
var myStrs = newStr.split(" ");
// Array(2) ["abc", "123"]
```

In the above, we split `newStr` into an array of two substrings, with `" "` as the separator. We will discuss JavaScript arrays shortly.

Note that we can call `.split("")`, we split the string into substrings where each substring is a character from the original string.

```
var myChars = newStr.split("");
// Array(7) [ "a", "b", "c", " ", "1", "2", "3" ]
```

When we call `.split()` without any argument the string is not splitted and the result array contains one element which is the whole string.

```
var notSplitted = newStr.split();
// Array [ "abc 123" ]
```

There are three different methods for extracting a sub-string from a string in JavaScript

```
myStr.substring(1, 3);
// "bc"
```

extracts the subtring starting from index `1` until `2`. The first argument is the starting index and the second one is the end index (end index itself is excluded in the output). The second method is `.substr` method:

```
myStr.substr(1, 2);
// "bc"
```

extracts `2` characters starting from index `1`. The first argument is the starting index and the second one is the length of the substring. The third method is using `.slice`:

```
myStr.slice(1, 3);
// "bc"
```

behaves the same as `.substring()`. However, when `.slice()` is given a single argument, it behaves like the index access.

```
myStr.slice(2) == myStr[2];
```

`.slice()` accepts negative integers as arguments.

```
myStr.slice(-3); // last three
// "abc"
myStr.slice(-2); // last two
// "bc"
myStr.slice(-3, -1); // the third last followed by the second last.
// "ab"
```

Array

In JavaScript, we use `[` and `]` to enclose a sequence of items to create an array.

```
var myArr = ["a", "b", "c"];
```

Similar to String, we may use `[idx]` to access an element of an array at position `idx`.

```
myArr[0];
// "a"
```

And the `.slice()` method also works on array.

```
myArr.slice(1, 3);
// Array ["b", "c"]
```

The `.reverse()` method revesrse the array in-place.

```
myArr.reverse();
myArr;
// Array ["c", "b", "a"]
```

Like string, `.concat()` method concatenates two arrays to produce a new one.

```
myArr.concat(["d"]);
// Array ["c", "b", "a", "d"]
```

Note that the `+` operator does not concatenate arrays.

```
myArr + ["d"];
// 'c,b,ad'
```

Method `join()` joins all the elements of an array back to a string, assuming all elements are string or can be converted to string.

```
myArray.join(",");
// "c,b,a"
```

Method `push()` inserts a new element at the end of the array and `pop()` removes the last element. These two elements turn an array into a stack data structure.

```
myArr.push("d");
myArr;
// Array ["c", "b", "a", "d"]
myArr.pop();
// "d"
myArr;
// Array ["c", "b", "a"]
```

To insert an element at the begining of an array, we may use the `.unshift()` method.

```
myArr.unshift("d");
myArr;
// Array ["d", "c", "b", "a"]
```

Note that `unshift` can insert more than one elements, e.g. `arr.unshift(e1, e2,...)`.

To insert an element at a particular position of an array, we use `.splice()` method.

```
myArr.splice(1, 0, "z");
myArr;
// Array ["d", "z", "c", "b", "a"]
```

In the above, we call `splice()` with three arguments. The first argument denotes the index position of the array that we would like to insert the new element. the second argument specifies how many arguments we would like to remove from this position onwards before the insertion. In this case we don't want to remove any existing element. The third argument denotes the new element that we would like to insert.

Without any surprise, we may also use `splice()` to remove elements from an array.

```
myArr.splice(1, 1);
myArr;
// Array ["d", "c", "b", "a"]
```

Object

Objects in JavaScritpts are key-value collection. It is very similar to the dictionary found in Python.

```
var myObj = { apple: 100, orange: 50 };
myObj["apple"];
// 100
myObj["durian"] = 200;
myObj;
// { 'apple': 100, 'orange': 50, 'durian': 200 }
```

Given a key of an object is of string type, besides using `obj[key]` to access the value, we may use the dot operator to access the value associated with the key.

```
myObj.durian;
// 200
```

Besides strings as keys, we can use numbers as keys in JavaScript objects, though it is uncommon.

```
myObj[1] = 1000;
myObj[1];
// 1000
```

However we cannot use dot operator to access values associated with number keys in JavaScript objects.

```
myObj.1; // syntax error
```

To remove a key from a JavaScript object, we may use the `delete` operator.

```
delete myObj[1];
myObj;
// { 'apple': 100, 'orange': 50, 'durian': 200 }
```

Given an object `o`, `Object.keys(o)` returns all the keys in this object and `Object.entries(o)` returns all the key-value pairs in the object.

```
Object.keys(myObj);
// ['apple', 'orange', 'durian']
Object.entries(myObj);
// [['apple', 100], ['orange', 50], ['durian', 200]]
```

Control flow statements

Like other general purpose language, JavaScript offers the following standard control flow statements.

if-else

```
var apple_count = null;
if ("apple" in myObj) {
  apple_count = myObj.apple;
} else {
  apple_count = 0;
}
```

In the above we assign the value associatd with the key `apple` in `myObj` if `apple` exists in `myObj`'s key set. Otherwise, `0` is assigned. As a side note, `null` denotes a null value in JavaScript. It means "not-exist".

for

```
var sum = 0;
var kvs = Object.entries(myObj);
for (let i = 0; i < kvs.length; i++) {
  sum = sum + kvs[i][1];
}
sum;
// 350
```

In the above snippet, we use a `for` loop to compute the sum of all values in the object `myObj`. First we initialize the `sum` variable to `0`. We compute the key-value entries in `myObj` and store them `kvs`. In the third statement, we use a for-loop it iterate through every key-value pair and extract the value. In the for statement, we make use of an index variable `i`. Starting from 0, `i` can be used to make reference to the `i+1`-th entry in the array `kvs`. At the end of each iteration we increment `i` by 1. The loop stops when `i == kvs.length`.

LOCAL SCOPE

Note that in the `for` statement we use `let` to declare a variable `i`. Remember the difference between `var` and `let` is as follows,

1. Variables declared with `var` is accessible in the entire function body.
2. Variables declared with `let` is accessible within the current block.

In the above example, `i` is not accessible before and after the for-loop.

Besides `var` and `let`, we can declare a constant variable using `const`

```
const pi = 3.14;
```

Constant variables are immutable, i.e. once assigned their values cannot be updated.

We could rewrite the for-loop in the following

```
for (let i in kvs) {
  sum = sum + kvs[i][1];
}
```

Notice that `i` is the keys of object `kvs`. `for-in` statement will loop through indices if the given iterable is an array.

Alternatively,

```
for (let kv of kvs) {
  sum = sum + kv[1];
}
```

In this case `kv` denotes the values from the array `kvs`. `for-of` statement will loop through each item if the given iterable is an array.

While

Besides for-loops, we can repeat operations with `while`.

```
var i = 0;
while (i < kvs.length) {
  sum = sum + kvs[i][1];
  i++;
}
```

forEach and map

Given an array, we can use `.forEach(p)` to apply function `p` to each element in the array.

```
kvs.forEach(function (kv) {
  sum = sum + kv[1];
});
```

In the above we pass anonymous function to add `kv[1]` to the `sum` variable. Note that the functio provided is a function without return statement.

If we want to apply a function `f` to each element of the array to produce a new array, we use `.map(f)`. The function provided here has a return statement.

```
var kvs2 = kvs.map(function (kv) {
  return [kv[0].toUpperCase(), kv[1]];
});
kvs2;
// [['APPLE', 100], ['ORANGE', 50], ['DURIAN', 200]]
```

Note that in JavaScript there is no tuple datatype.

JavaScript Operators

Binary Operators

Operators are predefined symbols in JavaScripts that allows us to compute result. For instance

```
var x = 3;
var y = 1;
var z = x + y; // z is 4
```

In the above context, this operator `+` sums up its given arguments since both are in number type.

```
typeof x; // number
typeof y; // number
```

The binary operator `+` is overloaded.

```
var a = "3";
var b = 1;
var c = a + b; // what is c?
```

In the above `+`'s left operand is a string and the right is a number, JavaScript performs some implicit type coercion to convert the second operand to string, hence `+` becomes a string concatenation operator.

Another commonly use binary operator in JavaScript is `==` which used for comparison.

```
var i = 100;
var isEven = i % 2 == 0; // isEven is true
```

What about?

```
var isEven2 = i % 2 == "0"; // isEven2 is true or false?
```

JavaScript coerces the right operand from string to number for us, hence `isEven2` is `true`

However there situations in which such an implicit type coercion is dangerous

```
var isEven3 = i % 2 == ""; // isEven3 is true??!?
```

To prevent this kind of situation, it's better to use a coercion-free comparison.

```
var isEven4 = i % 2 === ""; // isEven4 is false!
```

Things for you to check.

Please try to test out the following yourself, (try with different browsers).

1. Add a number to a `null` value.
2. Add a number to a `NaN` value.
3. Compare two string values.
4. Compare a number to a string value.

Unary Operators

Operators taking only one operand are called unary operators.

For instance, the `-` symbol is a unary operator as it takes a numerical operand and changes the sign of the operand.

The `!` symbol is another unary operator which takes boolean expression and return negation of the operand.

Class

In JavaScript, we can define a custom template for creating objects.

```
class Rect {
  constructor(w, h) {
    this.width = w;
    this.height = h;
  }
  area() {
    return this.width * this.height;
  }
}

var r = new Rect(10, 5);
console.log(r.area());
```

In the above code snippet, we define a class `Rect`. Every object instance of `Rect` has a `width` and a `height` attribute. The `constructor()` is invoked when we instantiate an object instance from a class via the `new` keyword. Besides the constructor method, the `Rect` class defines an `area()` method which computes the area of the object. To invoke a normal method, we use the `.` operator.

Inheritance (Only available in ES6 onwards)

Since JavaScript version ES6, we can define subclasses using the `extends` keyword.

```
class Square extends Rect {
  constructor(s) {
    super(s, s);
    this.side = s;
  }
}

var s = new Square(10);
console.log(s.area());
```

With inheritence objects instantiated from a subclass inherits methods and attributes belong to the base classes.

Most of the modern browsers support ES6. In case you need to handle some legacy browsers, or you would like understand how class and inheritance were compiled, you could read up the references

- JavaScript prototypes and prototype inheritance

```
https://www.freecodecamp.org/news/prototypes-and-inheritance-in-
javascript/#:~:text=In%20JavaScript%2C%20an%20object%20can,from%20other%20objects
```

Higher order functions

Functions as inputs

We have seen how functions are used in JavaScript in the previous unit. We consider some advanced usage of functions.

```
function incr(x) {
  return x + 1;
}

function twice(f, x) {
  return f(f(x));
}

console.log(twice(incr, 10)); // 12
```

In the above we define a function `incr` which increments a numerical input by `1`. Then we define a function `twice`, which takes a function argument `f` and another argument `x`. We apply `f` to `x` to produce an intermediate result then applies `f` again to the intermediate result.

`twice` is known as a higher order function as it takes another function as its input argument. Using higher order functions allows us to reuse codes.

```
function dbl(x) {
  return x * 2;
}

console.log(twice(dbl, 10)); // 40
```

Functions as a returned value (Closure)

Look at the code below and try to answer the question in comment:

```
function aFunc(a) {
  b = a * 5;
}
aFunc(2); // a=2, b=10

// Question: Is it possible to access or interact with the variables a and b
after the funcion is invoked above?
```

The answer is no. However, there is a way achieve it.

We can define a function that returns another function.

```
function part_sum(x) {
  return function (y) {
    return x + y;
  };
}

var incr_too = part_sum(1);
console.log(twice(incr_too, 10)); // 12
```

In the above, we define a partial sum function, `part_sum` which takes an input `x` and returns an anonymous function that expects input `y`. The inner function returns the sum of `x` and `y`. `incr_too` is a reference to the anonymous function produced by `part_sum(1)` that expects `y` and returns `1 + y`.

In this case, `part_sum` is a higher order function.

Closure

A closure in JavaScript is an inner function having access to its outer function's state/variables. In the example above, the inner function is a *closure* because the function is still having a reference to the variable `x` even after `part_sum(1)` has been invoked and returned. You can 'interact' with the variable `x` by invoking the `incr_too`.

Functions as call-backs

Function call-backs, also known as continuations, turn out to be very useful for us to define operations that should be performed in sequence. For instance,

```
var x = 1;
var y = incr(x);
var z = dbl(y);
console.log(z);
```

In the above we would like to increment `x` by `1` then multiply the intermediate result `y` by `2`. With call-backs, we could rewrite the above as

```
function incr_with_callback(x, f) {
  var r = incr(x);
  f(r);
}

function dbl_with_callback(y, g) {
  var r = dbl(y);
  g(r);
}
var x = 1;
incr_with_callback(x, function (y) {
  dbl_with_callback(y, function (z) {
    console.log(z);
  });
});
```

We introduce some variants of `incr` and `dbl` by calling the original functions to obtain the result. Instead of returning the results, we pass the results to the call back functions `f` (or `g`). `f` (and `g`) are known as the continuations, which means even when the main function terminates, it continues the execution by calling `f` (or `g`).

These two versions should produce the same result. One may argue that the second one is too hard to read and does not bring any significant value over the first version.

Application of Call-backs

Call-backs are widely used in JavaScript. One essential usage of call-back is to support asynchronous progamming.

Asynchronous Programming

Synchronous programming means instructions are executed one after another. On the other hand, asynchronous programming means instructions are not executed in the order they are structured. We need this "out-of-order" execution when there exists some long-running

instruction that occurs in the middle of the program, but we do not want this long-running instruction to block the rest of instructions. For example, while waiting for an API to return an output, or waiting for the query results from database.

One possible way to actualize asynchronous programming is to leverage multi-threading, i.e. to spawn a new thread from the main thread to execute the long-running instruction. We can use an analogy that a *thread* is like a virtual processor executing a task, so multi-threading allows you to start executing multiple task at the same time, altough not necessarily in parallel.

However, implementing asynchronous programming with multi-threading might lead to extra memory overhead because each thread needs to pre-allocate and maintain its own call stack space and size.

Single threaded Event-driven Programming

JavaScript takes a **different** approach in supporting asynchronous programming without using multi-threading.

From this point onwards, we narrow the definition of "callbacks" to be the functions that will be invoked when an event occurs.

Like many other languages, JavaScript executes a program with a call stack and a heap. Heap is a memory space for us to store non-primitive value such as objects, strings and arrays. For now our focus is on the call stack.

When a JavaScript starts, a `main` frame is added to the call stack.Global variables are stored in the `main` frame. When there is a function call, a new frame is placed on-top of the call stack. Local variables with primitive values are stored inside the frame. The frame is then popped when the function returns. The `main` frame is popped when all instructions in the JavaScript program has been executed. For example

```
function f(x) {
  var y = x + 1;
  return g(y);
}
function g(z) {
  return z * 2;
}
f(10);
```

In the above program, we define two functions `f` and `g`. The call stack begins with the folowing frame.

The main program call `f(10)`. We place a new frame on the call stack. When executing `f(10)`, we assign `10+1` to a local variable `y` in the call frame of `f(10)`.

Before returning the result, we call `g(y)` which is `g(11)` which creates another call frame. The following diagram shows the call stack with three frames when the program execution point is right before `return z*2`.

As `g(11)` call returns we remove the top most frame, i.e. `g(11)`, then back to the frame `f(10)`.

We return `22` as result and remove the second frame, `f(10)`.

Lastly, we remove the `main` frame as we reach to the end of the program.

Callback Queue

Things become more interesting in the presence of call-back functions associated with some UI elements. Recall one of our earlier examples,

```
1: function handleButton1Click() {
2:   var textbox1 = document.getElementById("textbox1");
3:   var span1 = document.getElementById("span1");
4:   span1.innerHTML = textbox1.value;
5: }
6: function run() {
7:   var button1 = document.getElementById("button1");
8:   button1.addEventListener("click", handleButton1Click);
9: }
10: document.addEventListener("DOMContentLoaded", run);
```

The program contains three portions, the two function definitions and a statement. JavaScript run-time in the browser executes them sequentially and synchronously, in the `main` call frame. To illustrate the step by step program execution let's consider the following table. The first column is the program counter. The second column captures the call stack, for simplicity, we use python's style list to represent a stack, where the left most element the bottom frame. The third column is a mapping from (UI) events to call-back functions, we use Python's style dictionary syntax. The fourth coulumn is the callback queue (again we use python's list style)

line num	call stack	event reg table	callback queue
1	[main]	{}	[]
6	[main]	{}	[]
10	[main]	{ `DomContentLoaded` : `run` }	[]

The run-time first execute the function declaration in line 1, in call frame `main`, which does not affect the event registration table nor the callback queue. Similar observation applies when it move on to execute the function declaration in line 6. When the program counter is at line 10, we are add the last statement of the main program. Add an event listener to the even `DOMContentLoaded`, which creates an entry to the even registration table, mapping `DOMContentLoaded` to the function `run` as callback. After line 10, the call stack becomes empty as `main` ended. However the JavaScript run-time for rendering this web page is still running. It is executing an event-loop, in which it periodically check whether there is anything is the callback queue. At this point in time the browser continues to render the HTML page. When the HTML document is fully rendered, the browser triggers an `DomContentLoaded` event. At this point the callback function `run` associated with the event is enqueued to the callback queue. Since the call stack is empty and the callback queue is not, the JavaScript run-time dequeues `run` from the callback queue and creates a call frame in the call stack. It continues to move the program counter from lines 7-9.

program counter (line num)	call stack	event reg table	callback queue
...	...	...	...
	[]	{ `DomContentLoaded` : `run` }	[`run`]
7	[`run`]	{ `DomContentLoaded` : `run` }	[]
8	[`run`]	{ `DomContentLoaded` : `run` }	[]
9	[`run`]	{ `DomContentLoaded` : `run`, `button1.click` : `handleButton1Click` }	[]

At the end of line 9, a new event (`button1` being clicked) is being registered, mapping to `handleButton1Click`. The call stack becomes empty again. The JavaScript run-time goes into the event loop again, waiting for next event to take place.

When someone clicks on the button `button1`, the callback function `handleButton1Click` will be enqeued into the callback queue. The run-time will "move" it into the call stack to execute it.

We have purposely ignored something ...

Actually, besides the callback, there are something called *promises* which enable JavaScript code to support asynchronous programming. We will discuss it in details in the next class.

Cohort Exercise

1. (Not Graded) Add some `console.log()` to the `textandbutton.js` and open `textandbutton.html` to observe in what order the call backs are executed.
2. (Graded) Complete the following HTML and JavaScript codes, so that when the button `Submit` is clicked, the min and the max of the sequence of numbers (seperated by ",") entered in the input text box will be displayed in the `span` elements.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="minmax.js"></script>
    <meta charset="utf-8" />
```

```
    <title>50.003 sample code: Min and Max</title>
  </head>
  <body>
    <div>Your input: <input id="textbox1" type="text" /></div>
    <div>Min: <span id="min"></span></div>
    <div>Max: <span id="max"></span></div>
    <div><button id="button1">Submit</button></div>
  </body>
</html>
```

```
function numbers(l) {
  var o = [];
  for (let i in l) {
    var n = parseInt(l[i], 10);
    if (!isNaN(n)) {
      o.push(n);
    }
  }
  return o;
}
// input: an array of numbers
// output: an object containing 'min', with the minimum of the array and 'max'
the maximum of the array.
function min_max(a) {
  var min = null;
  var max = null;
  // TODO: fixme
  return { min: min, max: max };
}

function handleButton1Click() {
  var textbox1 = document.getElementById("textbox1");
  var min = document.getElementById("min");
  var max = document.getElementById("max");
  var items = textbox1.value.split(",");
  var obj = min_max(numbers(items));
  min.innerHTML = obj["min"];
  max.innerHTML = obj["max"];
}

function run() {
  var button1 = document.getElementById("button1");
  // TODO: fixme
}

document.addEventListener("DOMContentLoaded", run);
```