

COMP1531



Development - Advanced Functions

Lecture 4.1

Author(s): Hayden Smith



[\(Download as PDF\)](#)

In This Lecture

- **Why?** 🤔
 - Higher level languages have many powerful methods of how we can use functions
- **What?** 📄
 - Function Syntax
 - First-class Functions
 - Higher Order Functions
 - Callbacks



Function Syntax

In Javascript there are three equivalent ways to define functions.

Method 1

```
1 function sum(a, b) {  
2   return a + b;  
3 }
```

fn_syntax_1.js

Method 2

```
1 const sum = function(a, b) {  
2   return a + b;  
3 };
```

fn_syntax_2.js

Method 3

```
1 const sum = (a, b) => {  
2   return a + b;  
3 };
```

fn_syntax_3.js

- Method 1 is the old school method. Though it's fine :)
- Method 2 and 3 are similar, treating functions like variables.
- Method 3 is more modern and has other advantages.



Function Syntax

In Javascript there are three equivalent ways to define functions.

Method 1

```
1 function sum(a: number, b: number) {  
2   return a + b;  
3 }
```

fn_syntax_1.ts

Method 2

```
1 const sum = function(a: number, b: number) {  
2   return a + b;  
3 };
```

fn_syntax_2.ts

Method 3

```
1 const sum = (a: number, b: number) => {  
2   return a + b;  
3 };
```

fn_syntax_3.ts

- Method 1 is the old school method. Though it's fine :)
- Method 2 and 3 are similar, treating functions like variables.
- Method 3 is more modern and has other advantages.



Function Syntax

Method 3 also has another advantage. IFF (If and only if) the function body is a single line body that simply returns a value, the braces { } and return keyword can be omitted.

Method 3

```
1 const sum = (a: number, b: number) => {  
2   return a + b;  
3 };
```

[fn_syntax_3.ts](#)

Method 3 Compact

```
1 const sum = (a: number, b: number) => a + b;  
2  
3 // so short!!
```

[fn_syntax_3_compact.ts](#)



Function Syntax

Another example: You might have previously written this function like this:

Many String

```
1 function manyString(repeat: number, str: string) {  
2   let outString = '';  
3   for (let i = 0; i < repeat; i++) {  
4     outString += str;  
5   }  
6   return outString;  
7 }  
8 console.log(manyString(5, 'hello '));
```

many_string_1.ts



Function Syntax

But we can also write it like this

Many String

```
1 function manyString(repeat: number, str: string) {  
2   let outString = '';  
3   for (let i = 0; i < repeat; i++) {  
4     outString += str;  
5   }  
6   return outString;  
7 }  
8 console.log(manyString(5, 'hello '));
```

[many_string_1.ts](#)

Many String Update 1

```
1 const manyString = function(repeat: number, str: string) {  
2   let outString = '';  
3   for (let i = 0; i < repeat; i++) {  
4     outString += str;  
5   }  
6   return outString;  
7 };  
8 console.log(manyString(5, 'hello '));
```

[many_string_2.ts](#)



Function Syntax

Taking it a step further!

```
1 const manyString = function(repeat: number, str: string) {
2   let outString = '';
3   for (let i = 0; i < repeat; i++) {
4     outString += str;
5   }
6   return outString;
7 };
8 console.log(manyString(5, 'hello '));
```

many_string_2.ts

```
1 const manyString = (repeat: number, str: string) => {
2   let outString = '';
3   for (let i = 0; i < repeat; i++) {
4     outString += str;
5   }
6   return outString;
7 };
8 console.log(manyString(5, 'hello '));
```

many_string_3.ts



First-Class Functions

Variable Definition

```
1 const name = 'Hayden';  
2 console.log(name);
```

Function Definition

```
1 const getName = () => {  
2   return 'Hayden';  
3 };  
4 console.log(getName);
```

What do these different function syntaxes teach us about functions in Javascript?



First-Class Functions

Variable Definition

```
1 const name = 'Hayden';  
2 console.log(name);
```

Function Definition

```
1 const getName = () => {  
2   return 'Hayden';  
3 };  
4 console.log(getName);
```

What do these different function syntaxes teach us about functions in Javascript?

That we can treat functions similar to variables! E.G. What we're doing with the
`console.log`



First-Class Functions

Variable Definition

```
1 const name = 'Hayden';  
2 console.log(name);
```

Function Definition

```
1 const getName = () => {  
2   return 'Hayden';  
3 };  
4 console.log(getName);
```

What do these different function syntaxes teach us about functions in Javascript?

That we can treat functions similar to variables! E.G. What we're doing with the
`console.log`

Therefore in Javascript, we say the language has **first-class functions**



First Class Functions

A language has **first-class functions** when functions are treated just like any other variable.

Most notably, functions can be passed into functions just like variables can.



First Class Functions

You're already used to the idea that you can write a function that produces a different output based on the variable inputted.

```
1 const sayHi = (name: string) => {  
2   return `Hello ${name}!`;  
3 };  
4 console.log(sayHi('Hayden'));
```

fcf_var.ts



First-Class Functions

So let's expand that idea and say we can write a function that produces a different output based on a **function** that is inputted. We essentially pass the function object in, and then **call** the function inside the body.

```
1 type Fmtr = (str: string) => string;
2
3 function brackets(str: string) {
4   return `(${str})`;
5 }
6
7 function fullstop(str: string) {
8   return `${str}.`;
9 }
10
11 function sayHi(name: string, format: Fmtr) {
12   return `Hello, ${format(name)}!`;
13 }
14
15 const result = sayHi('Hayden', brackets) +
16               ' -- ' +
17               sayHi('Hayden', fullstop);
18
19 console.log(result);
```

[fcf_format_atomic.ts](#)



First-Class Functions

Now let's use an alternative syntax!

```
1 type Fmtr = (str: string) => string;
2
3 function brackets(str: string) {
4   return `${str}`;
5 }
6
7 function fullstop(str: string) {
8   return `${str}.`;
9 }
10
11 function sayHi(name: string, format: Fmtr) {
12   return `Hello, ${format(name)}!`;
13 }
14
15 const result = sayHi('Hayden', brackets) +
16   ' -- ' +
17   sayHi('Hayden', fullstop);
18
19 console.log(result);
```

[fcf_format_atomic.ts](#)

```
1 type Fmtr = (str: string) => string;
2
3 const brackets = (str: string) => `${str}`;
4 const fullstop = (str: string) => `${str}.`;
5 const sayHi = (name: string, format: Fmtr) => `Hello, ${format(name)}!`;
6
7 const result = sayHi('Hayden', brackets) +
8   ' -- ' +
9   sayHi('Hayden', fullstop);
10
11 console.log(result);
```

[fcf_format_atomic_alt.ts](#)



First-Class Functions

For something more complicated: Now let's use functions as arguments, but apply it within a loop.

```
1 type Fmtr = (str: string) => string;
2
3 function brackets(str: string) {
4   return `${str}`;
5 }
6
7 function lowercase(str: string) {
8   return str.toLowerCase();
9 }
10
11 const names = ['Hayden', 'Giuliana', 'Tam'];
12
13 function formatNames(list: string[], format: Fmtr) {
14   const newList = [];
15   for (const name of list) {
16     newList.push(format(name));
17   }
18   return newList;
19 }
20
21 const newName = formatNames(names, brackets);
22 console.log(newName);
23
24 const lowercaseNames = formatNames(names, lowercase);
25 console.log(lowercaseNames);
```

[fcf_format_loop.ts](#)



First-Class Functions

Comparing with new function syntax.

```
1 type Fmtr = (str: string) => string;
2
3 function brackets(str: string) {
4   return `${str}`;
5 }
6
7 function lowercase(str: string) {
8   return str.toLowerCase();
9 }
10
11 const names = ['Hayden', 'Giuliana', 'Tam'];
12
13 function formatNames(list: string[], format: Fmtr) {
14   const newList = [];
15   for (const name of list) {
16     newList.push(format(name));
17   }
18   return newList;
19 }
20
21 const newNames = formatNames(names, brackets);
22 console.log(newNames);
23
24 const lowercaseNames = formatNames(names, lowercase);
25 console.log(lowercaseNames);
```

fcf_format_loop.ts

```
1 type Fmtr = (str: string) => string;
2
3 const names = ['Hayden', 'Giuliana', 'Tam'];
4
5 function formatNames(list: string[], format: Fmtr) {
6   const newList: string[] = [];
7   for (const name of list) {
8     newList.push(format(name));
9   }
10  return newList;
11 }
12
13 console.log(formatNames(names, (str: string) => `${str}`));
14 console.log(formatNames(names, (str: string) => str.toLowerCase()));
15 console.log(formatNames(names, (str: string) => `[${str}]`));
```

fcf_format_loop_new.ts



First-Class Functions

Anonymous Functions

If we only intend to use a function once, we can pass it directly into a function. Because this function doesn't have a name, we call it an anonymous function.

```
1 type Fmtr = (str: string) => string;
2
3 const names = ['Hayden', 'Giuliana', 'Tam'];
4
5 function formatNames(list: string[], format: Fmtr) {
6   const newList: string[] = [];
7   for (const name of list) {
8     newList.push(format(name));
9   }
10  return newList;
11 }
12
13 console.log(formatNames(names, (str: string) => `${str}`));
14 console.log(formatNames(names, (str: string) => str.toLowerCase()));
15 console.log(formatNames(names, (str: string) => `[${str}]`));
```

[fcf_format_loop_new.ts](#)

```
1 type Fmtr = (str: string) => string;
2
3 const names = ['Hayden', 'Giuliana', 'Tam'];
4
5 function formatNames(list: string[], format: Fmtr) {
6   const newList = [];
7   for (const name of list) {
8     newList.push(format(name));
9   }
10  return newList;
11 }
12
13 const newNames = formatNames(names, (str) => {
14   return `${str}`;
15 });
16
17 console.log(newNames);
```

[fcf_format_loop_anon.ts](#)



First-Class Functions

In Summary: What?

First-class functions are predominately used in terms of letting **functions take in other functions as arguments**

In Summary: Why?

Allows us to create more concise and clear code. In particular, the use of anonymous functions for one-off usage make code both more compact and more readable. It's also the fundamental part of understanding callbacks.



First-Class Functions

In Summary: What?

First-class functions are predominately used in terms of letting **functions take in other functions as arguments**

In Summary: Why?

Allows us to create more concise and clear code. In particular, the use of anonymous functions for one-off usage make code both more compact and more readable. It's also the fundamental part of understanding callbacks.

Wait, what the %@# is a callback!?!?



Map, Reduce, Filter

Map, reduce, filter are functions that act on array objects that help us accomplish basic iterative tasks without the overhead of a loop setup. They are based on ideas of **first-class functions** and **anonymous functions**

- **Map:** Modify an array
- **Filter:** Select from an array
- **Reduce:** Summarise an array



Map, Reduce, Filter

Map

Takes an array of size N , and produces a new array of size N having modified each element according to a function passed in.

Same array size. Modified elements.



Map, Reduce, Filter

Map

Classic

```
1 const tutors = [  
2   'Simon',  
3   'Teresa',  
4   'Kaiqi',  
5   'Michelle',  
6 ];  
7  
8 const loudTutors = tutors.map((str: string)  
9   return `${str.toUpperCase()}!!!`;  
10 });  
11  
12 console.log(loudTutors);
```

map_old.ts

Modern

```
1 const tutors = [  
2   'Simon',  
3   'Teresa',  
4   'Kaiqi',  
5   'Michelle',  
6 ];  
7  
8 const shout = function(str: string) {  
9   return `${str.toUpperCase()}!!!`;  
10 };  
11  
12 const newList = tutors.map(shout);  
13 console.log(newList);
```

map.ts

Same array size. Modified elements.



Map, Reduce, Filter

Map

Classic

```
1 const tutors = [  
2   'Simon',  
3   'Teresa',  
4   'Kaiqi',  
5   'Michelle',  
6 ];  
7  
8 const loudTutors = tutors.map((str: string)  
9   return `${str.toUpperCase()}!!!`;  
10 });  
11  
12 console.log(loudTutors);
```

map_old.ts

Modern

```
1 const tutors = [  
2   'Simon',  
3   'Teresa',  
4   'Kaiqi',  
5   'Michelle',  
6 ];  
7  
8 const shout = function(str: string) {  
9   return `${str.toUpperCase()}!!!`;  
10 };  
11  
12 const newList = tutors.map(shout);  
13 console.log(newList);
```

map.ts

Same array size. Modified elements.



Map, Reduce, Filter

Filter

Takes an array of size N , and produces a new array of size $0..N$ without modifying any of the data.

Possibly smaller array size. Elements unchanged.



Filter, Reduce, Filter

Filter

Classic

Modern

```
1 type Student = {
2   name: string;
3   mark: number;
4 }
5
6 const students = [
7   {
8     name: 'Hayden',
9     mark: 45,
10  },
11  {
12    name: 'Yuchao',
13    mark: 75,
14  },
15  {
16    name: 'Giuliana',
17    mark: 85,
18  },
19  {
20    name: 'Tam',
21    mark: 85,
22  },
23 ];
24
25 const newList = students.map((student: Student) => student.mark >= 50 ? student : null);
26
27 console.log(newList);
```

filter_old.ts

```
1 const marks = [65, 72, 81, 40, 56];
2
3 const isPass = function(mark: number) {
4   return mark >= 50;
5 };
6
7 const newList = marks.filter(isPass);
8 console.log(newList);
```

filter.ts

Possibly smaller array size. Elements unchanged.



Filter, Reduce, Filter

Filter

Classic

Modern

```
1 type Student = {
2   name: string;
3   mark: number;
4 }
5
6 const students = [
7   {
8     name: 'Hayden',
9     mark: 45,
10  },
11  {
12    name: 'Yuchao',
13    mark: 75,
14  },
15  {
16    name: 'Giuliana',
17    mark: 85,
18  },
19  {
20    name: 'Tam',
21    mark: 85,
22  },
23 ];
24
25 const newList = students.map((student: Student) => student.mark >= 50 ? student : null);
26
27 console.log(newList);
```

filter_old.ts

```
1 const marks = [65, 72, 81, 40, 56];
2
3 const isPass = function(mark: number) {
4   return mark >= 50;
5 };
6
7 const newList = marks.filter(isPass);
8 console.log(newList);
```

filter.ts

Possibly smaller array size. Elements unchanged.



Map, Reduce, Filter

Reduce

Executes a reducer function (that you provide) on each member of the array resulting in a single output value.

Array turned into atomic value.



Filter, Reduce, Filter

Reduce

Classic

```
1 const students = [  
2   { name: 'Amy', mark: 55 },  
3   { name: 'Bob', mark: 43 },  
4   { name: 'Cap', mark: 34 },  
5   { name: 'Dex', mark: 23 },  
6 ];  
7  
8 let single = 0;  
9 for (const student of students) {  
10   single += student.mark;  
11 }  
12 console.log(single);
```

[reduce_old.ts](#)

Modern

```
1 type Student = {  
2   name: string;  
3   mark: number;  
4 }  
5  
6 const students: Student[] = [  
7   { name: 'Amy', mark: 55 },  
8   { name: 'Bob', mark: 43 },  
9   { name: 'Cap', mark: 34 },  
10  { name: 'Dex', mark: 23 },  
11 ];  
12  
13 const sum = (prev: number, curr: Student) => {  
14   return prev + curr.mark;  
15 };  
16  
17 const single = students.reduce(sum, 0);  
18 console.log(single);
```

[reduce.ts](#)

Array turned into atomic value.



Map, Reduce, Filter

Anonymous Functions

We can use anonymous functions to further streamline our previous code.

```
1 const tutors = ['Simon', 'Teresa', 'Kaiqi', 'Michelle'];
2 const newList = tutors.map((string) => `${string.toUpperCase()}!!!`);
3 console.log(newList);
```

[map_stream.ts](#)

```
1 const marks = [65, 72, 81, 40, 56];
2 const newList = marks.filter((mark) => mark >= 50);
3 console.log(newList);
```

[filter_stream.ts](#)

```
1 const students = [
2   { name: 'Amy', mark: 55 },
3   { name: 'Bob', mark: 43 },
4   { name: 'Cap', mark: 34 },
5   { name: 'Dex', mark: 23 },
6 ];
7 const single = students.reduce((prev, curr) => prev + curr.mark, 0);
8 console.log(single);
```

[reduce_stream.ts](#)



Map, Reduce, Filter Combined

We can combine these methods to create some very clean code.

Print the average mark of those student's who passed the 60 mark exam. Express the average as a %.

```
1 const marks = [39, 43.2, 48.6, 24, 33.6];
2 console.log(marks);
3 const normalisedMarks = marks.map(m => 100 * m / 60);
4 console.log(normalisedMarks);
5 const passingMarks = normalisedMarks.filter(m => m >= 50);
6 console.log(passingMarks);
7 const total = passingMarks.reduce((a, b) => a + b, 0);
8 console.log(total);
9 const average = total / passingMarks.length;
10 console.log(average);
```

[mapfilterreduce.ts](#)



Higher Order Functions

Higher-order Functions are essentially functions that return functions. Think of them like mini function factories.

Let's go through a code cleanup journey.

```
1 function congratMarkPS(name: string) {
2   return `Congratulations ${name} on your pass`;
3 }
4 function congratMarkCR(name: string) {
5   return `Congratulations ${name} on your credit`;
6 }
7 function congratMarkDN(name: string) {
8   return `Congratulations ${name} on your distinction`;
9 }
10 console.log(congratMarkCR('Hayden'));
```

hoc_1.ts



Higher Order Functions

```
1 function congratMarkPS(name: string) {
2   return `Congratulations ${name} on your pass`;
3 }
4 function congratMarkCR(name: string) {
5   return `Congratulations ${name} on your credit`;
6 }
7 function congratMarkDN(name: string) {
8   return `Congratulations ${name} on your distinction`;
9 }
10 console.log(congratMarkCR('Hayden'));
```

hoc_1.ts

```
1 function congratWrapper(markstr: string, name: string) {
2   return `Congratulations ${name} on your ${markstr}`;
3 }
4 function congratMarkPS(name: string) {
5   return congratWrapper('pass', name);
6 }
7 function congratMarkCR(name: string) {
8   return congratWrapper('credit', name);
9 }
10 function congratMarkDN(name: string) {
11   return congratWrapper('distinction', name);
12 }
13 console.log(congratMarkCR('Hayden'));
```

hoc_2.ts



Higher Order Functions

```
1 function congratWrapper(markstr: string, name: string) {
2   return `Congratulations ${name} on your ${markstr}`;
3 }
4 function congratMarkPS(name: string) {
5   return congratWrapper('pass', name);
6 }
7 function congratMarkCR(name: string) {
8   return congratWrapper('credit', name);
9 }
10 function congratMarkDN(name: string) {
11   return congratWrapper('distinction', name);
12 }
13 console.log(congratMarkCR('Hayden'));
```

hoc_2.ts

```
1 const congratWrapper = (markstr: string, name: string) => {
2   return `Congratulations ${name} on your ${markstr}`;
3 };
4 const congratMarkPS = (name: string) => {
5   return congratWrapper('pass', name);
6 };
7 const congratMarkCR = (name: string) => {
8   return congratWrapper('credit', name);
9 };
10 const congratMarkDN = (name: string) => {
11   return congratWrapper('distinction', name);
12 };
13 console.log(congratMarkCR('Hayden'));
```

hoc_3.ts



Higher Order Functions

```
1 const congratWrapper = (markstr: string, name: string) => {
2   return `Congratulations ${name} on your ${markstr}`;
3 };
4 const congratMarkPS = (name: string) => {
5   return congratWrapper('pass', name);
6 };
7 const congratMarkCR = (name: string) => {
8   return congratWrapper('credit', name);
9 };
10 const congratMarkDN = (name: string) => {
11   return congratWrapper('distinction', name);
12 };
13 console.log(congratMarkCR('Hayden'));
```

hoc_3.ts

```
1 function genCongratMark(markstr: string) {
2   const ret = function(name: string) {
3     return `Congratulations ${name} on your ${markstr}`;
4   };
5   return ret;
6 }
7 const congratMarkPS = genCongratMark('pass');
8 const congratMarkCR = genCongratMark('credit');
9 const congratMarkDN = genCongratMark('distinction');
10
11 console.log(congratMarkCR('Hayden'));
```

hoc_4.ts



Higher Order Functions

```
1 function genCongratMark(markstr: string) {
2   const ret = function(name: string) {
3     return `Congratulations ${name} on your ${markstr}`;
4   };
5   return ret;
6 }
7 const congratMarkPS = genCongratMark('pass');
8 const congratMarkCR = genCongratMark('credit');
9 const congratMarkDN = genCongratMark('distinction');
10
11 console.log(congratMarkCR('Hayden'));
```

[hoc_4.ts](#)

```
1 const genCongratMark = (markstr: string) => {
2   const ret = (name: string) => {
3     return `Congratulations ${name} on your ${markstr}`;
4   };
5   return ret;
6 };
7
8 const congratMarkPS = genCongratMark('pass');
9 const congratMarkCR = genCongratMark('credit');
10 const congratMarkDN = genCongratMark('distinction');
11
12 console.log(congratMarkCR('Hayden'));
```

[hoc_5.ts](#)



Higher Order Functions

```
1 const genCongratMark = (markstr: string) => {
2   const ret = (name: string) => {
3     return `Congratulations ${name} on your ${markstr}`;
4   };
5   return ret;
6 };
7
8 const congratMarkPS = genCongratMark('pass');
9 const congratMarkCR = genCongratMark('credit');
10 const congratMarkDN = genCongratMark('distinction');
11
12 console.log(congratMarkCR('Hayden'));
```

[hoc_5.ts](#)

```
1 const genCongratMark = (markstr: string) => {
2   return (name: string) => {
3     return `Congratulations ${name} on your ${markstr}`;
4   };
5 };
6
7 const congratMarkPS = genCongratMark('pass');
8 const congratMarkCR = genCongratMark('credit');
9 const congratMarkDN = genCongratMark('distinction');
10
11 console.log(congratMarkCR('Hayden'));
```

[hoc_6.ts](#)



Higher Order Functions

```
1 const genCongratMark = (markstr: string) => {
2   return (name: string) => {
3     return `Congratulations ${name} on your ${markstr}`;
4   };
5 };
6
7 const congratMarkPS = genCongratMark('pass');
8 const congratMarkCR = genCongratMark('credit');
9 const congratMarkDN = genCongratMark('distinction');
10
11 console.log(congratMarkCR('Hayden'));
```

[hoc_6.ts](#)

```
1 const genCongratMark = (markstr: string) => (name: string) =>
2   `Congratulations ${name} on your ${markstr}`;
3
4 const congratMarkPS = genCongratMark('pass');
5 const congratMarkCR = genCongratMark('credit');
6 const congratMarkDN = genCongratMark('distinction');
7
8 console.log(congratMarkCR('Hayden'));
```

[hoc_7.ts](#)

Higher Order Functions

Summary

Higher order functions are syntactically sleek ways to generalise function definitions.

Feedback



Or go to the [form here](#).

