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 (HOCs)
  - 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 }
```

4.1_fn_syntax_1.js

Method 2

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

4.1_fn_syntax_2.js

Method 3

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

4.1_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

```ts
function sum(a: number, b: number) {
  return a + b;
}
```

4.1_fn_syntax_1.ts

Method 2

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

4.1_fn_syntax_2.ts

Method 3

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

4.1_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

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

4.1_fn_syntax_3.ts

Method 3 Compact

```ts
const sum = (a: number, b: number) => a + b;

// so short!!
```

4.1_fn_syntax_3_compact.ts

🔠 Function Syntax

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

Many String

```ts
function manyString(repeat: number, str: string) {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
}
console.log(manyString(5, 'hello '));
```

4.1_many_string_1.ts

🔠 Function Syntax

But we can also write it like this

Many String

```ts
function manyString(repeat: number, str: string) {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
}
console.log(manyString(5, 'hello '));
```

4.1_many_string_1.ts

Many String Update 1

```ts
const manyString = function(repeat: number, str: string) {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
};
console.log(manyString(5, 'hello '));
```

4.1_many_string_2.ts

🔠 Function Syntax

Taking it a step further!

```ts
const manyString = function(repeat: number, str: string) {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
};
console.log(manyString(5, 'hello '));
```

4.1_many_string_2.ts

```ts
const manyString = (repeat: number, str: string) => {
  let outString = '';
  for (let i = 0; i < repeat; i++) {
    outString += str;
  }
  return outString;
};
console.log(manyString(5, 'hello '));
```

4.1_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'));
```

4.1_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.

```
type Fmtr = (str: string) => string;

function brackets(str: string) {
  return `(${str})`;
}

function fullstop(str: string) {
  return `${str}.`;
}

function sayHi(name: string, format: Fmtr) {
  return `Hello, ${format(name)}!`;
}

const result = sayHi('Hayden', brackets) +
               ' -- ' +
               sayHi('Hayden', fullstop);

console.log(result);
```

4.1_fcf_format_atomic.ts

👭 First-Class Functions

Now let's use an alternative syntax!

```ts
type Fmtr = (str: string) => string;

function brackets(str: string) {
  return `(${str})`;
}

function fullstop(str: string) {
  return `${str}.`;
}

function sayHi(name: string, format: Fmtr) {
  return `Hello, ${format(name)}!`;
}

const result = sayHi('Hayden', brackets) +
               ' -- ' +
               sayHi('Hayden', fullstop);

console.log(result);
```

4.1_fcf_format_atomic.ts

```ts
type Fmtr = (str: string) => string;

const brackets = (str: string) => `(${str})`;
const fullstop = (str: string) => `${str}.`;
const sayHi = (name: string, format: Fmtr) => `Hello, ${format(name)}

const result = sayHi('Hayden', brackets) +
               ' -- ' +
               sayHi('Hayden', fullstop);

console.log(result);
```

4.1_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.

```
type Fmtr = (str: string) => string;

function brackets(str: string) {
  return `(${str})`;
}

function lowercase(str: string) {
  return str.toLowerCase();
}

const names = ['Hayden', 'Giuliana', 'Tam'];

function formatNames(list: string[], format: Fmtr) {
  const newList = [];
  for (const name of list) {
    newList.push(format(name));
  }
  return newList;
}
```

```
20
21 const newNames = formatNames(names, brackets);
22 console.log(newNames);
23
24 const lowercaseNames = formatNames(names, lowercase);
25 console.log(lowercaseNames);
```

4.1_fcf_format_loop.ts

👫 First-Class Functions

Comparing with new function syntax.

```ts
type Fmtr = (str: string) => string;

function brackets(str: string) {
  return `(${str})`;
}

function lowercase(str: string) {
  return str.toLowerCase();
}

const names = ['Hayden', 'Giuliana', 'Tam'];

function formatNames(list: string[], format: Fmtr
  const newList = [];
  for (const name of list) {
    newList.push(format(name));
  }
  return newList;
}

const newNames = formatNames(names, brackets);
console.log(newNames);

const lowercaseNames = formatNames(names, lowerca
console.log(lowercaseNames);
```

4.1_fcf_format_loop.ts

```ts
type Fmtr = (str: string) => string;

const names = ['Hayden', 'Giuliana', 'Tam'];

function formatNames(list: string[], format: Fmtr) {
  const newList: string[] = [];
  for (const name of list) {
    newList.push(format(name));
  }
  return newList;
}

console.log(formatNames(names, (str: string) => `(${str})`));
console.log(formatNames(names, (str: string) => str.toLowerCas
console.log(formatNames(names, (str: string) => `[${str}]`));
```

4.1_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.

```ts
type Fmtr = (str: string) => string;

const names = ['Hayden', 'Giuliana', 'Tam'];

function formatNames(list: string[], format: Fmtr) {
  const newList: string[] = [];
  for (const name of list) {
    newList.push(format(name));
  }
  return newList;
}

console.log(formatNames(names, (str: string) => `(${str})`));
console.log(formatNames(names, (str: string) => str.toLowerCas
console.log(formatNames(names, (str: string) => `[${str}]`));
```

4.1_fcf_format_loop_new.ts

```ts
type Fmtr = (str: string) => string;

const names = ['Hayden', 'Giuliana', 'Tam'];

function formatNames(list: string[], format: Fmt
  const newList = [];
  for (const name of list) {
    newList.push(format(name));
  }
  return newList;
}

const newNames = formatNames(names, (str) => {
  return `(${str})`;
});

console.log(newNames);
```

4.1_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

```
const tutors = [
  'Simon',
  'Teresa',
  'Kaiqi',
  'Michelle',
];

const loudTutors = tutors.map((str: string)
  return `${str.toUpperCase()}!!!`;
});

console.log(loudTutors);
```

4.1_map_old.ts

Modern

```
const tutors = [
  'Simon',
  'Teresa',
  'Kaiqi',
  'Michelle',
];

const shout = function(str: string) {
  return `${str.toUpperCase()}!!!`;
};

const newList = tutors.map(shout);
console.log(newList);
```

4.1_map.ts

Same array size. Modified elements.

🧪 Map, Reduce, Filter

Map

Classic

```ts
const tutors = [
  'Simon',
  'Teresa',
  'Kaiqi',
  'Michelle',
];

const loudTutors = tutors.map((str: string)
  return `${str.toUpperCase()}!!!`;
});

console.log(loudTutors);
```

4.1_map_old.ts

Modern

```ts
const tutors = [
  'Simon',
  'Teresa',
  'Kaiqi',
  'Michelle',
];

const shout = function(str: string) {
  return `${str.toUpperCase()}!!!`;
};

const newList = tutors.map(shout);
console.log(newList);
```

4.1_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

```ts
type Student = {
  name: string;
  mark: number;
}

const students = [
  {
    name: 'Hayden',
    mark: 45,
  },
  {
    name: 'Yuchao',
    mark: 75,
  },
  {
    name: 'Giuliana',
    mark: 85,
  },
  {
    name: 'Tam',
    mark: 85,
  },
];

const newList = students.map((student: Student) => student.mark >

```

Modern

```ts
const marks = [65, 72,

const isPass = function
  return mark >= 50;
};

const newList = marks.f
console.log(newList);
```

4.1_filter.ts

```
27 console.log(newList);
```

4.1_filter_old.ts

Possibly smaller array size. Elements unchanged.

Filter, Reduce, Filter

Filter

Classic

```ts
type Student = {
  name: string;
  mark: number;
}

const students = [
  {
    name: 'Hayden',
    mark: 45,
  },
  {
    name: 'Yuchao',
    mark: 75,
  },
  {
    name: 'Giuliana',
    mark: 85,
  },
  {
    name: 'Tam',
    mark: 85,
  },
];

const newList = students.map((student: Student) => student.mark >

```

Modern

```ts
const marks = [65, 72,

const isPass = function
  return mark >= 50;
};

const newList = marks.f
console.log(newList);
```

4.1_filter.ts

```
27 console.log(newList);
```

4.1_filter_old.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

```ts
const students = [
  { name: 'Amy', mark: 55 },
  { name: 'Bob', mark: 43 },
  { name: 'Cap', mark: 34 },
  { name: 'Dex', mark: 23 },
];

let single = 0;
for (const student of students) {
  single += student.mark;
}
console.log(single);
```

4.1_reduce_old.ts

Modern

```ts
type Student = {
  name: string;
  mark: number;
}

const students: Student[] = [
  { name: 'Amy', mark: 55 },
  { name: 'Bob', mark: 43 },
  { name: 'Cap', mark: 34 },
  { name: 'Dex', mark: 23 },
];

const sum = (prev: number, curr: Student) => {
  return prev + curr.mark;
};

const single = students.reduce(sum, 0);
console.log(single);
```

4.1_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);
```

4.1_map_stream.ts

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

4.1_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);
```

4.1_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);
```

4.1_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'));
```

4.1_hoc_1.ts

🙏 Higher Order Functions

```ts
function congratMarkPS(name: string) {
  return `Congratulations ${name} on your pass`;
}
function congratMarkCR(name: string) {
  return `Congratulations ${name} on your credit`;
}
function congratMarkDN(name: string) {
  return `Congratulations ${name} on your distinction`;
}
console.log(congratMarkCR('Hayden'));
```

4.1_hoc_1.ts

```ts
function congratWrapper(markstr: string, name: string) {
  return `Congratulations ${name} on your ${markstr}`;
}
function congratMarkPS(name: string) {
  return congratWrapper('pass', name);
}
function congratMarkCR(name: string) {
  return congratWrapper('credit', name);
}
function congratMarkDN(name: string) {
  return congratWrapper('distinction', name);
}
console.log(congratMarkCR('Hayden'));
```

4.1_hoc_2.ts

🙏 Higher Order Functions

```ts
function congratWrapper(markstr: string, name: string) {
  return `Congratulations ${name} on your ${markstr}`;
}
function congratMarkPS(name: string) {
  return congratWrapper('pass', name);
}
function congratMarkCR(name: string) {
  return congratWrapper('credit', name);
}
function congratMarkDN(name: string) {
  return congratWrapper('distinction', name);
}
console.log(congratMarkCR('Hayden'));
```

4.1_hoc_2.ts

```ts
const congratWrapper = (markstr: string, name: string) => {
  return `Congratulations ${name} on your ${markstr}`;
};
const congratMarkPS = (name: string) => {
  return congratWrapper('pass', name);
};
const congratMarkCR = (name: string) => {
  return congratWrapper('credit', name);
};
const congratMarkDN = (name: string) => {
  return congratWrapper('distinction', name);
};
console.log(congratMarkCR('Hayden'));
```

4.1_hoc_3.ts

🙏🏾 Higher Order Functions

```ts
const congratWrapper = (markstr: string, name: string) => {
  return `Congratulations ${name} on your ${markstr}`;
};
const congratMarkPS = (name: string) => {
  return congratWrapper('pass', name);
};
const congratMarkCR = (name: string) => {
  return congratWrapper('credit', name);
};
const congratMarkDN = (name: string) => {
  return congratWrapper('distinction', name);
};
console.log(congratMarkCR('Hayden'));
```

4.1_hoc_3.ts

```ts
function genCongratMark(markstr: string) {
  const ret = function(name: string) {
    return `Congratulations ${name} on your ${markstr}`;
  };
  return ret;
}
const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_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'));
```

4.1_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'));
```

4.1_hoc_5.ts

🙏🏾 Higher Order Functions

```ts
const genCongratMark = (markstr: string) => {
  const ret = (name: string) => {
    return `Congratulations ${name} on your ${markstr}`;
  };
  return ret;
};

const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_hoc_5.ts

```ts
const genCongratMark = (markstr: string) => {
  return (name: string) => {
    return `Congratulations ${name} on your ${markstr}`;
  };
};

const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_hoc_6.ts

🙏🏾 Higher Order Functions

```ts
const genCongratMark = (markstr: string) => {
  return (name: string) => {
    return `Congratulations ${name} on your ${markstr}`;
  };
};

const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_hoc_6.ts

```ts
const genCongratMark = (markstr: string) => (name: string) =>
  `Congratulations ${name} on your ${markstr}`;

const congratMarkPS = genCongratMark('pass');
const congratMarkCR = genCongratMark('credit');
const congratMarkDN = genCongratMark('distinction');

console.log(congratMarkCR('Hayden'));
```

4.1_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.