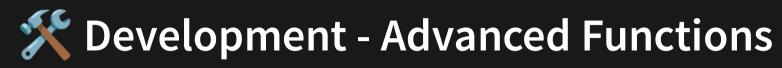# COMP1531

🛠️ 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

```
1 function sum(a: number, b: number) {
2   return a + b;
3 }
```
4.1_fn_syntax_1.ts

Method 2

```
1 const sum = function(a: number, b: number) {
2   return a + b;
3 };
```
4.1_fn_syntax_2.ts

Method 3

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

5

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 };
```

4.1_fn_syntax_3.ts

Method 3 Compact

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

4.1_fn_syntax_3_compact.ts

🔤 Function Syntax

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

Many String

```typescript
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 '));
```

4.1_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 '));
```

4.1_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 '));
```

4.1_many_string_2.ts

Function Syntax

Taking it a step further!

```typescript
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 '));
```

4.1_many_string_2.ts

```typescript
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 '));
```

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.

```typescript
const sayHi = (name: string) => {
  return `Hello ${name}!`;
};
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.

```typescript
 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);
```

4.1_fcf_format_atomic.ts

👫 First-Class Functions

Now let's use an alternative syntax!

```typescript
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);
```

4.1_fcf_format_atomic.ts

```typescript
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);
```

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.

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

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

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);
```

4.1_fcf_format_loop.ts

👫 First-Class Functions

Comparing with new function syntax.

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

4.1_fcf_format_loop.ts

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

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.

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

4.1_fcf_format_loop_new.ts

```typescript
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);
```

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

```ts
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 = [];
13 for (const tutor of tutors) {
14   const newTutor = shout(tutor);
15   newList.push(newTutor);
16 }
17 console.log(newList);
```

4.1_map_old.ts

Modern

```ts
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);
```

4.1_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 shout = function(str: string) {
 9   return `${str.toUpperCase()}!!!`;
10 };
11
12 const newList = [];
13 for (const tutor of tutors) {
14   const newTutor = shout(tutor);
15   newList.push(newTutor);
16 }
17 console.log(newList);
```

4.1_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);
```
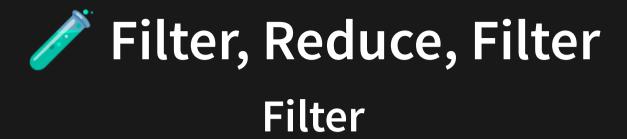
4.1_map.ts

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.

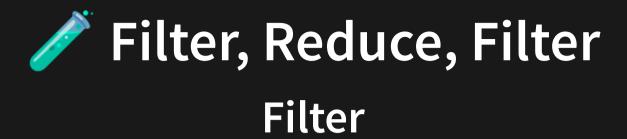**Possibly smaller array size. Elements unchanged.**

🧪 Filter, Reduce, Filter

Filter

Classic

```
 1 const marks = [65, 72, 81, 40, 56];
 2
 3 const isPass = function(mark: number) {
 4   return mark >= 50;
 5 };
 6
 7 const newList = [];
 8 for (const mark of marks) {
 9   if (isPass(mark)) {
10     newList.push(mark);
11   }
12 }
13 console.log(newList);
```

4.1_filter_old.ts

Modern

```
 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);
```

4.1_filter.ts

Possibly smaller array size. Elements unchanged.

🧪 Filter, Reduce, Filter
Filter

Classic

```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 = [];
8  for (const mark of marks) {
9    if (isPass(mark)) {
10     newList.push(mark);
11   }
12 }
13 console.log(newList);
```

4.1_filter_old.ts

Modern

```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);
```

4.1_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.
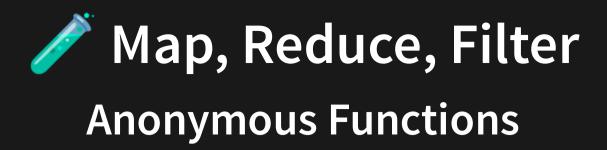
Array turned into atomic value.

🧪 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);
```

4.1_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);
```

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.

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

4.1_map_stream.ts

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

4.1_filter_stream.ts

```typescript
const students = [
  { name: 'Amy', mark: 55 },
  { name: 'Bob', mark: 43 },
  { name: 'Cap', mark: 34 },
  { name: 'Dex', mark: 23 },
];
const single = students.reduce((prev, curr) => prev + curr.mark, 0);
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 %.

```typescript
const marks = [39, 43.2, 48.6, 24, 33.6];
const normalisedMarks = marks.map(m => 100 * m / 60);
const passingMarks = normalisedMarks.filter(m => m >= 50);
const total = passingMarks.reduce((a, b) => a + b, 0);
const average = total / passingMarks.length;
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.

```typescript
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

🙏 Higher Order Functions

```typescript
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

```typescript
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'));
```

4.1_hoc_2.ts

🙏 Higher Order Functions

```typescript
 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'));
```

4.1_hoc_2.ts

```typescript
 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'));
```

4.1_hoc_3.ts

🙏 Higher Order Functions

```typescript
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'));
```

4.1_hoc_3.ts

```typescript
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

🙏 Higher Order Functions

```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'));
```

4.1_hoc_4.ts

```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

```typescript
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

```typescript
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'));
```

4.1_hoc_6.ts

🙏 Higher Order Functions

```typescript
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'));
```

4.1_hoc_6.ts

```typescript
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'));
```

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.