Recursion I

Week 3: Day 10

April 2021

Overview

- What does a function return?
- Functions calling other functions
- Call stack
- Recursion
  - Countdown
  - Factorial

What does a function return: 1

```
function funcA(num) {
  console.log(num);
}
```

```
let funcReturn = funcA(5);
>5
console.log(funcReturn)
>undefined
```

- If no explicit ‘return’, function returns ‘undefined’

What does a function return: 2

```
function funcA(num) {
  return;
  console.log(num);
}
```

```
let funcReturn = funcA(5);
>
console.log(funcReturn)
>undefined
```

- function doesn’t execute code after hitting ‘return’
- Just ‘return’ still returns ‘undefined’ value

What does a function return: 3

```
function funcA(num) {
  return num-1;
  console.log(num);
}
```

```
let funcReturn = funcA(5);
>
console.log(funcReturn)
>4
```

- ‘return’ will return whatever value follows

Functions calling functions

```
function funcA(num) {
  console.log('entering funcA', num);
  let retFuncB = funcB(num - 1);
  console.log('leaving funcA);
  return num * retFuncB;
}

function funcB(num) {
  console.log('inside funcB', num);
  return num;
}
```

```
let funcReturn = aFunc(5);
>entering funcA 5
>inside funcB 4
>leaving funcA
console.log(funcReturn)
>20
```

- funcA will wait till funcB returns before resuming rest of code
- funcA can capture funcB return and do stuff with it

Call Stack

```
function funcA(num) {
  console.log('entering funcA', num);
  let retFuncB = funcB(num - 1);
  console.log('leaving funcA');
  return num * retFuncB;
}

function funcB(num) {
  console.log('inside funcB', num);
  return num;
}
```

```
let funcReturn = funcA(5);
console.log(funcReturn);
>entering funcA 5
```

call stack	input	return value
aFunc	5	5 * retFuncB

- Call stack keeps track of current function at the top

Call Stack

```
function funcA(num) {
  console.log('entering funcA', num);
  let retFuncB = funcB(num - 1);
  console.log('leaving funcA');
  return num * retFuncB;
}

function funcB(num) {
  console.log('inside funcB', num);
  return num;
}
```

```
let funcReturn = funcA(5);
console.log(funcReturn);
>entering funcA 5
```

call stack	input	return value
bFunc	4	num
aFunc	5	5 * retFuncB

- Call stack keeps track of previous functions that are not completed. It stacks current function on top of previous function

Call Stack

```
function funcA(num) {
  console.log('entering funcA', num);
  let retFuncB = funcB(num - 1);
  console.log('leaving funcA');
  return num * retFuncB;
}

function funcB(num) {
  console.log('inside funcB', num);
  return num;
}
```

```
let funcReturn = funcA(5);
console.log(funcReturn);
>entering funcA 5
>inside funcB
```

call stack	input	return value
bFunc	4	=> 4
aFunc	5	5 * retFuncB

Call Stack

```
function funcA(num) {
  console.log('entering funcA', num);
  let retFuncB = funcB(num - 1);
  console.log('leaving funcA');
  return num * retFuncB;
}

function funcB(num) {
  console.log('inside funcB', num);
  return num;
}
```

```
let funcReturn = funcA(5);
console.log(funcReturn);
>entering funcA 5
>inside funcB
```

call stack	input	return value
aFunc	5	=> 5 * 4

- After current function finishes execution, it is taken out of call stack

Call Stack

```
function funcA(num) {
  console.log('entering funcA', num);
  let retFuncB = funcB(num - 1);
  console.log('leaving funcA');
  return num * retFuncB;
}

function funcB(num) {
  console.log('inside funcB', num);
  return num;
}
```

```
let funcReturn = funcA(5);
console.log(funcReturn);
>entering funcA 5
>inside funcB
>leaving funcA
```

call stack	input	return value
aFunc	5	=> 20

Call Stack

```
function funcA(num) {
  console.log('entering funcA', num);
  let retFuncB = funcB(num - 1);
  console.log('leaving funcA');
  return num * retFuncB;
}

function funcB(num) {
  console.log('inside funcB', num);
  return num;
}
```

```
let funcReturn = funcA(5);
console.log(funcReturn);
>entering funcA 5
>inside funcB
>leaving funcA
>20
```

call stack	input	return value

- Call stack is empty after outermost function exits

Recursion

- IS a function calling itself
- Any problems using loops can also be solved by using Recursion
    - Recursion can be more readable
    - Recursion uses more memory than loops

Setting up recursion

1. Base case
    a. The most basic case
    b. Base case does not call itself
2. Recursive case
    a. Recursive case calls itself
    b. Input passed to function must change so that you will eventually trigger the base case!

Recursion - countdown

```
function funcA(num) {
  //BASE CASE
  if(num < 1) return 0;

  //RECURSIVE CASE
  console.log(num);
  let ret = funcA(num-1);
  return num + ret
}
```

```
let funcReturn = funcA(4);
console.log(funcReturn);
>10
```

call stack	input	return
funcA	0	0
funcA	1	1 + ret
funcA	2	2 + ret
funcA	3	3 + ret
funcA	4	4 + ret

- Call stack keeps piling self calls till BASE CASE
- Then starts unwinding downwards

Recursion - factorial

```
function funcA(num) {
  //BASE CASE
  if(num <= 0) return 1
  if(num === 1) return 1

  //RECURSIVE CASE
  let ret = funcA(num-1);
  return ret * num;
}
```

```
let funcReturn = funcA(5);
console.log(funcReturn);
>120
```

call stack	input	return
funcA	1	1
funcA	2	ret * 2
funcA	3	ret * 3
funcA	4	ret * 4
funcA	5	ret * 5

- Call stack keeps piling self calls till BASE CASE
- Then starts unwinding downwards

Overview

- What does a function return?
- Functions calling other functions
- Call stack
- Recursion
  - Countdown
  - Factorial