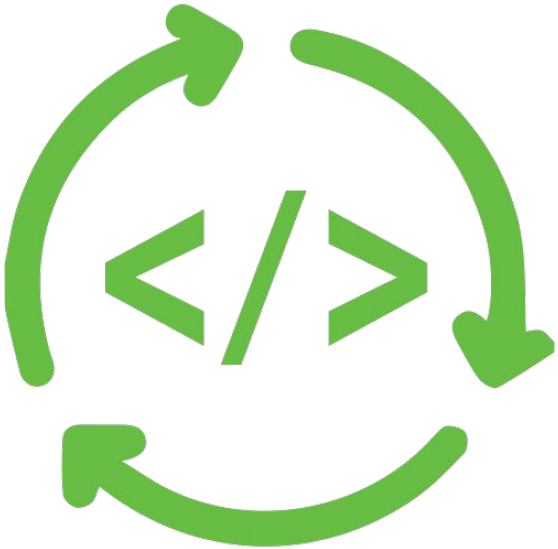
Lesson 10: Functions

How to compartmentalize your code!

Download the Lesson 10 file, unzip, and drag it to your desktop.

How can we control what our file does?

- Let's say we want to accomplish a bunch of different tasks in the same file, but want to use these tasks at different times
  - What could we do?
- Without explicitly teaching it, we've been using **functions**
  - Main way to make code reusable
- No more "node main.js **1**"
- We directly control what runs.



Function Definition Syntax

- Similar to other statements, except functions are **named.**
  - Use something descriptive!
- Declare with the keyword “function”
- Name the function
- List **parameters**
- Content of the function contained in braces
- This along does not make the function run!

```
function example(parameter1, parameter2) {
  console.log(parameter1);
  //prints value of parameter1
  console.log(parameter2);
  //prints value of parameter2
}
```

Function Invocation

- So far, we've treated functions like tasks
  - That's the right intuition
  - Different execution from now on
- In order to get a function to run, we need to **invoke** it.
  - We do this by calling the function by name, and giving it inputs for its parameters
  - Anything within the braces will execute

```
print("Hello World"); //function invocation

function print(input) { //function definition
  console.log(input);
}
```

Function Parameters

- Order matters!
  - Need to make sure the invocation inputs match up with the definition parameters
- Pay attention to your types
  - If you define the function, make sure your invocation inputs match
  - If someone else defines the function read the **documentation**
- Documentation lets you know what the function does, and what the parameters are

```
//Math.pow(x,y) Example
Math.pow(2,3); //8
Math.pow(3,2); //9
```

Task #1

Your turn!

- Open up Lesson10/main.js in Atom
  - Follow the instructions
- Take 10 minutes - get as far as you can
- Send questions in the chat
- You need to invoke the task1 function yourself! No more command line arguments.

Function Reusability

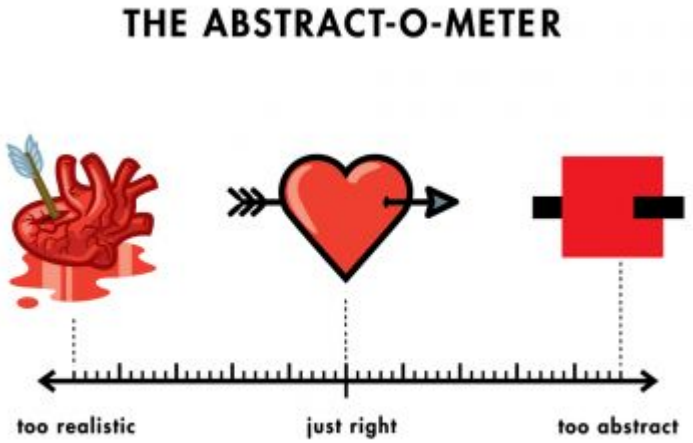
- You can call the same function as many times as you want!
- You can change the inputs as necessary
- Helpful function names allow us to think more intuitively about our programs

```
let words = ["Hi", "my", "name", "is", "Max"];
for(let i = 0; i<words.length; i++){
  print(words[i]); //this will print each element individually.
}

function print(input) { //function definition
  console.log(input);
}
```

Abstraction

- Another way of saying “simplify”
- Functions allow us to think “abstractly” when solving problems
- Building functions for complex, but isolated tasks make it easier to tie together a solution



Example

- Back to factorials
  - Code to solve factorials is complicated
  - But isolated!
  - We can make this a function
- Having a reusable function allows us to “abstract” away all the complicated code

```
let nums = [5,2,7,4];
for(let i = 0; i<nums.length; i++) {
  nums[i] = factorial(nums[i]);
}
console.log(nums);


function factorial(number) {
  let result = 1;
  for(number; number>1; number--) {
    result = result * number;
  }
  return result;
}
```

Task #2

Now it’s time to write a function from scratch!

- Follow the instructions
- Take 10 minutes - get as far as you can
- Send questions in the chat
- Remember to change the function you invoke at the top of the file.

Review

What have we learned?

- Functions allow us to reuse the same code
  - Definition - state what the function does
  - Invocation - tell the program to execute the function
- Parameters
  - Gives the function values
  - Order of parameters must match between the definition and all invocations
  - Beware of expected types!

Wrap-Up

Tomorrow is our last class!

Email questions to info.codedelaware@gmail.com

See you at 6:30!