11.1. Functions as Values

Functions are powerful tools in any programming language, and JavaScript uses these tools in some flexible and creative ways. This chapter introduces a bit more of the power of functions.

11.1.1. Functions Are Data

We defined a value as “a specific piece of data.” Some examples are the number `42`, the string `"LC101"`, and the array `["MO", "FL", "DC"]`. *Functions are also values*, and while they appear to be very different from other values we have worked with, they share many core characteristics.

In particular, functions have a data type, just like all other values. Recall that a **data type** is a group of values that share characteristics, such as strings and numbers. Strings share the characteristice of having a length, while numbers don’t. Numbers can be manipulated in ways that strings cannot, via operations like division and subtraction.

Example

The data type of the type conversion function `Number` is `function`. In fact, all functions are of type `function`.

```
console.log(typeof 42);
console.log(typeof "LC101");
console.log(typeof Number);
```

Console Output

```
number
string
function
```

Like other data types, functions may be assigned to variables. If we create a function named `hello` we can assign it to a variable with this syntax:

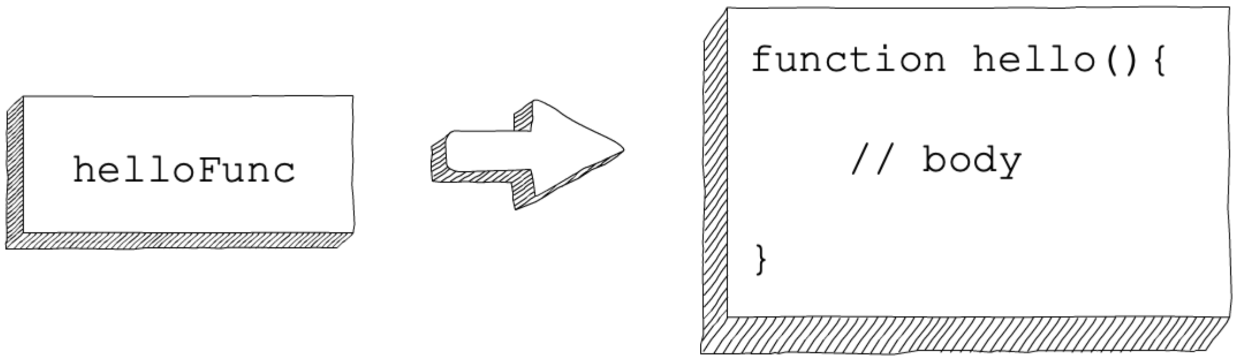
```
function hello() {

   // function body

}

let helloFunc = hello;
```

When a variable refers to a function, we can use the variable name to *call* the function:

```
helloFunc();
```

The variable `helloFunc` can be thought of as an *alias* for the function `hello`. When we call the function `helloFunc`, JavaScript sees that it refers to the function `hello` and calls that *specific* function.

When we use a variable *name*, we are really using its *value*. If the variable `class` is assigned the value `"LC101"`, then `console.log(class)` prints `"LC101"`. When a variable holds a function, it behaves the same way as when it holds a number or a string. The variable *refers to* the function.



A variable that refers to a function.

Again, *functions are values*. They can be used just like general values. For example:

- Functions may be assigned to variables.
- Functions may be used in expressions, such as comparisons.
- Functions may be converted to other data types.
- Functions may be printed using `console.log`.
- Functions may be passed as arguments to other functions.
- Functions may be returned from other functions.

Some of these function behaviors do not prove to be useful. You will probably never need to convert a function to a boolean, or ask whether a function is greater than 5. However, other behaviors, like passing functions as arguments and assigning them to variables, turn out to be *extremely* useful.

11.2. Anonymous Functions

You already know one method for creating a function:

```
function myFunction(parameter1, parameter2,..., parameterN) {

   // function body

}
```

A function defined in this way is a **named function** (`myFunction`, in the example above).

Many programming languages, including JavaScript, allow us to create **anonymous functions**, which *do not* have names. We can create an anonymous function by simply leaving off the function name when defining it:

```
function (parameter1, parameter2,..., parameterN) {

   // function body

}
```

You might be asking yourself, *How do I call a function if it doesn't have a name?!* Good question. Let's address that now.

11.2.1. Anonymous Function Variables

Anonymous functions are often assigned to variables when they are created, which allows them to be called using the variable's name.

Example

Let's create and use a simple anonymous function that returns the sum of two numbers.

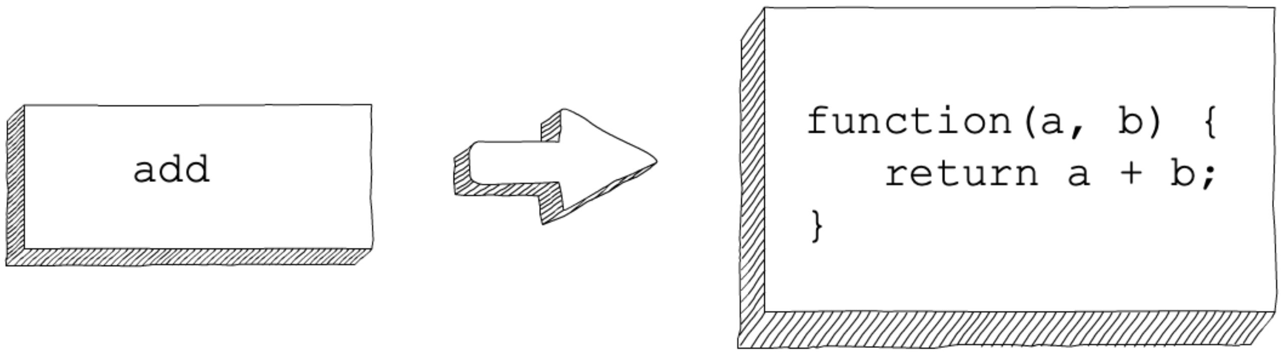
```
1 let add = function(a, b) {
2    return a + b;
3 };
4
5 console.log(add(1, 1));
```

Console Output

```
2
```

The variable `add` refers to the anonymous function created on lines 1 through 3. We call the function using the *variable*name, since the function doesn't have a name.

The visual analogy here is the same as that of a variable referring to a named function.



A variable that refers to an anonymous function.

Warning

Like other variable declarations, an assignment statement using an anonymous function should be terminated by a semi-colon, `;`. This is easy to overlook, since named functions do *not* end with a semi-colon.

11.2.2. Check Your Understanding

Question

Convert the following named function to an anonymous function that is stored in a variable.

```
function reverse(str) {
   let lettersArray = str.split('');
   let reversedLettersArray = lettersArray.reverse();
   return reversedLettersArray.join('');
}
```

repl.it

```
let reversed = function(str) {
  let lettersArray = str.split('');
  let reversedLettersArray = lettersArray.reverse();
  return reversedLettersArray.join('');
}
console.log(reversed("LaunchCode"));
```

Question

Consider the code sample below, which declares an anonymous function beginning on line 1.

```
1  let f1 = function(str) {
2     return str + str;
3  };
4
5  let f2 = f1;
```

Which of the following are valid ways of invoking the anonymous function with the argument `"abcd"`? (Choose all that apply.)

a. `f1("abcd");`
b. `function("abcd");`
c. `f2("abcd");`
d. It is not possible to invoke the anonymous function, since it doesn't have a name.

Question

Complete the following code snippet so that it logs an error message if `userInput` is negative.

```
let logger = function(errorMsg) {
   console.log("ERROR: " + errorMsg);
};

if (userInput < 0) {
   ____________("Invalid input");
}
```

repl.it

```
let logger = function(errorMsg) {
   console.log("ERROR: " + errorMsg);
};

userInput = -3;
if (userInput < 0) {
   logger("Invalid input");
}
```

11.3. Passing Functions as Arguments

Functions are data, and therefore can be passed around just like other values. This means a function can be *passed to another function* as an argument. This allows the *function being called* to use the *function argument* to carry out its action. This turns out to be extremely useful.

Examples best illustrate this technique, so let’s look at a couple now.

11.3.1. Example: `setTimeout`

The built-in function `setTimeout` allows a programmer to pass a function, specifying that it should be called at later point in time. Its basic syntax is:

```
setTimeout(func, delayInMilliseconds);
```

Example

Suppose we want to log a message with a 5 second delay. Since five seconds is 5000 milliseconds (1 second = 1000 milliseconds), we can do so like this:

```
function printMessage() {
   console.log("The future is now!");
}

setTimeout(printMessage, 5000);
```

repl.it

Console Output

```
"The future is now!"
```

Try It!

Is the call to `printMessage` actually delayed? Don’t just take our word for it, try this yourself. Play with our example to change the delay.
The function `printMessage` is *passed* to `setTimeout` the same as any other argument.

A common twist often used by JavaScript programmers is to use an *anonymous* function as an argument.

Example

This program has the same behavior as the one above. Instead of creating a named function and passing it to `setTimeout`, it creates an anonymous function within `setTimeout`’s argument list.

```
setTimeout(function () {
   console.log("The future is now!");
}, 5000);
```

Examples like this look odd at first. However, they become easier to read over time. Additionally, code that passes anonymous functions is ubiquitous in JavaScript.

11.3.2. Example: The Array Method `map`

The array method `map` allows for every element in an array to be *mapped* or *translated*, using a given function. Here's how to use it:

```
let mappedArray = someArray.map(func);
```

The argument `func` should take a single value from the array and return a new value. The returned array, `mappedArray`, contains each of the individual return values from the mapping function, `func`.

Example

```
let nums = [3.14, 42, 4811];

let timesTwo = function (n) {
   return n*2;
};

let doubled = nums.map(timesTwo);

console.log(nums);
console.log(doubled);
```

Console Output

```
[3.14, 42, 4811]
[ 6.28, 84, 9622 ]
```

Notice that `map` does *not* alter the original array.

When using `map`, many programmers will define the mapping function anonymously in the same statement as the method call `map`.

Example

This program has the same output as the one immediately above. The mapping function is defined anonymously within the call to `map`.

```
let nums = [3.14, 42, 4811];

let doubled = nums.map(function (n) {
   return n*2;
});

console.log(doubled);
```

Console Output

```
[ 6.28, 84, 9622 ]
```

11.3.3. Check Your Understanding

Question

Similar to the `map` example above, finish the program below to halve each number in an array.

```
let nums = [3.14, 42, 4811];

// TODO: Write a mapping function
// and pass it to .map()
let halved = nums.map();

console.log(halved);
```

repl.it

```
let nums = [3.14, 42, 4811];

// TODO: Write a mapping function
// and pass it to .map()
let halved = nums.map(function (n) {
   return n/2;
});

console.log(halved);
```

Question

Use the `map` method to map an array of strings. For each name in the array, map it to the first initial.

```
let names = ["Chris", "Jim", "Sally", "Blake", "Paul"];

// TODO: Write a mapping function
// and pass it to .map()
let firstInitials = names.map();

console.log(firstInitials);
```

repl.it

```
let names = ["Chris", "Jim", "Sally", "Blake"];

// TODO: Write a mapping function
// and pass it to .map()
let firstInitials = names.map(function(n) {
  return n.slice(0,1);
});

console.log(firstInitials);
```

11.4. Receiving Function Arguments

The previous section illustrates how a function can be passed to another function as an argument. This section takes the opposite perspective to *write* functions that can take other functions as arguments.

11.4.1. Example: A Generic Input Validator

Our first example will be a generic input validator. It will prompt a user for input, using a parameter to the function to do the actual work of validating the input.

Example

```
const input = require('readline-sync');

function getValidInput(prompt, isValid) {

   // Prompt the user, using the prompt string that was passed
   let userInput = input.question(prompt);

   // Call the boolean function isValid to check the input
   while (!isValid(userInput)) {
      console.log("Invalid input. Try again.");
      userInput = input.question(prompt);
   }

   return userInput;
}

// A boolean function for validating input
let isEven = function(n) {
   return Number(n) % 2 === 0;
};

console.log(getValidInput('Enter an even number:', isEven));
```

Sample Output

```
Enter an even number: 3
Invalid input. Try again.
Enter an even number: 5
Invalid input. Try again.
Enter an even number: 4
4
```

The function `getValidInput` handles the work of interacting with the user, while allowing the validation logic to be customized. This separates the different concerns of validation and user interaction, sticking to the idea that *a function should do only one thing*. It also enables more reusable code. If we need to get different input from the user, we can simply call `getValidInput` with different arguments.

Example

This example uses the same `getValidInput` function defined above with a different prompt and validator function. In this case, we check that a potential password has at least 8 characters.

```javascript
const input = require('readline-sync');

function getValidInput(prompt, isValid) {

   let userInput = input.question(prompt);

   while (!isValid(userInput)) {
      console.log("Invalid input. Try again.");
      userInput = input.question(prompt);
   }

   return userInput;
}

let isValidPassword = function(password) {

   // Passwords should have at least 8 characters
   if (password.length < 8) {
      return false;
   }

   return true;
};

console.log(getValidInput('Create a password:', isValidPassword));
```

Sample Output

```
Create a password: launch
Invalid input. Try again.
Create a password: code
Invalid input. Try again.
Create a password: launchcode
launchcode
```

Try It!

1. Use our `getValidInput` function to ensure user input starts with “a”.
2. Create another validator that ensures user input is a vowel.

Try it at repl.it

11.4.2. Example: A Logger

Another common example of a function using another function to customize its behavior is that of logging. Real-world applications are capable of logging messages such as errors, warnings, and statuses. Such applications allow for log messages to be sent to one or more destinations. For example, the application may log messages to both the console and to a file.

We can write a logging function that relies on a function parameter to determine the logging destination.

11.4.2.1. A Simple Logger

Example

The `logError` function outputs a standardized error message to a location determined by the parameter `logger`.

```
   let fileLogger = function(msg) {
 1
 2   // Put the message in a file
 3
 4}
 5
 6function logError(msg, logger) {
 7   let errorMsg = 'ERROR: ' + msg;
 8   logger(errorMsg);
 9}
10
11logError('Something broke!', fileLogger);
12
```

Let's examine this example in more detail.

There are three main program components:

1. Lines 1-5 define `fileLogger`, which takes a string argument, `msg`. We have not discussed writing to a file, but Node.js is capable of doing so.
2. Lines 7-10 define `logError`. The first parameter is the message to be logged. The second parameter is the logging function that will do the work of sending the message somewhere. `logError` doesn't know the details of how the message will be logged. It simply formats the message, and calls `logger`.
3. Line 12 logs an error using the `fileLogger`.

This is the flow of execution:

1. `logError` is called, with a message and the logging function `fileLogger` passed as arguments.
2. `logError` runs, passing the constructed message to `logger`, which refers to `fileLogger`.
3. `fileLogger` executes, sending the message to a file.

11.4.2.2. A More Complex Logger

This example can be made even more powerful by enabling multiple loggers.

Example

The call to `logError` will log the message to both the console and a file.

```
 1let fileLogger = function(msg) {
 2
 3   // Put the message in a file
 4
 5}
 6
 7let consoleLogger = function(msg) {
 8
 9   console.log(msg);
10
11}
12
13function logError(msg, loggers) {
14
15   let errorMsg = 'ERROR: ' + msg;
16
17   for (let i = 0; i < loggers.length; i++) {
      logger[i](errorMsg);
   }

}

logError('Something broke!', [fileLogger, consoleLogger]);
```

The main change to the program is that `logError` now accepts an *array* of functions. It loops through the array, calling each logger with the message string.

As with the validation example, these programs separate behaviors in a way that makes the code more flexible. To add or remove a logging destination, we can simply change the way that we call `logError`. The code *inside* `logError` doesn't know how each logging function does it's job. It is concerned only with creating the message string and passing it to the logger(s).

11.4.3. A Word of Caution

What happens if a function expects an argument to be a function, but it isn't?

Try It!

```
function callMe(func) {
   func();
}

callMe("Al");
```

repl.it

Question

What type of error occurs when attempting to use a value that is NOT a function as if it were one?

11.5. Why Use Anonymous Functions?

At this point, you may be asking yourself *Why am I learning anonymous functions?* They seem strange, and their utility may not be immediately obvious. While the opinions of programmers differ, there are two main reasons why *we* think anonymous functions are important to understand.

11.5.1. Anonymous Functions Can Be Single-Use

There are many situations in which you will need to create a function that will only be used once. To see this, recall one of our earlier examples.

Example

The anonymous function created in this example cannot be used outside of `setTimeout`.

```
setTimeout(function () {
   console.log("The future is now!");
}, 5000);
```

Defining an anonymous function at the same time it is passed as an argument prevents it from being used elsewhere in the program.

Additionally, in programs that use lots of functions—such as web applications, as you will soon learn—defining functions anonymously, and directy within a function call, can reduce the number of names you need to create.

11.5.2. Anonymous Functions Are Ubiquitous in JavaScript

JavaScript programmers use anonymous functions *a lot*. Many programmers use anonymous functions with the same gusto as that friend of yours who puts hot sauce on *everything*.

Just because an anonymous function isn't needed to solve a problem doesn't mean that it *shouldn't* be used to solve the problem. Avoiding JavaScript code that uses anonymous functions is impossible.

Any programming problem in JavaScript can be solved *without* using anonymous functions. Thus, the extent to which you use them in your own code is somewhat a matter of taste. We will take the middle road throughout the rest of this course, regularly using both anonymous and named functions.

11.5.3. Check Your Understanding

Question

Explain the difference between named and anonymous functions, including an example of how an anonymous function can be used.

11.6. Recursion

11.6.1. Quick Review

In the previous chapter, we learned how to define a function and its parameters.

Example

```
function addTwoToNumber(num){
   return num += 2;
}

console.log(addTwoToNumber(12));
```

Console Output

```
14
```

When called, the parameter `num` is passed an argument, which in this case is the number `12`. The function executes and returns the value `14`, which the `console.log` statement prints.

11.6.1.1. Functions Can Call Other Functions

Functions should only accomplish one (preferably simple) task. To solve more complicated tasks, one small function must call other functions.

Example

```
function addTwoToNumber(num){
   return num += 2;
}

function addFiveToNumber(value){
   let result = addTwoToNumber(value) + 3;
   return result;
}

console.log(addFiveToNumber(12))
```

Console Output

```
17
```

Of course, there is no need to write a function to add 5 to a value, but the example demonstrates calling a function from within another function.

11.6.2. What Is Recursion?

In programming, the *divide and conquer* strategy solves a problem by breaking it down into smaller, simpler pieces. If these pieces *can all be solved in exactly the same way*, then we gain an additional advantage. Solving the big problem becomes a process of completing and combining the smaller parts.

Splitting up a large task into smaller, identical pieces allows us to reuse a single function rather than coding several different functions. We accomplish this by either:

a. Setting up a loop to call one function lots of times, OR
b. Building a function that splits up the large problem for us, until a *simplest case* is found and solved.

Recursion is the process of solving a larger problem by breaking it into smaller pieces that *can all be solved in exactly the same way*. The clever idea behind recursion is that instead of using a loop, a function simply calls *itself* over and over again, with each step reducing the size of the problem.

Through recursion, a problem eventually gets reduced to a very simple task, which can be immediately solved. This small answer sets up the solution for the previous step, which in turn solves the next bigger step. Properly built, the function combines all of the small answers to solve the original problem.

Many new programmers (and even veteran ones) find recursion an abstract and tricky concept. One helpful way to approach the idea is to walk through an example.

11.7. Recursion Walkthrough: The Base Case

To ease into the concept of recursion, let’s start with a loop task.

In the Arrays chapter, we examined the join method, which combines the elements of an array into a single string. If we have `arr = ['L', 'C', '1', '0', '1']`, then `arr.join('')` returns the string `'LC101'`.

We can reproduce this action with either a `for` or a `while` loop.

11.7.1. Joining Array Elements With a Loop

Examine the code samples below:

Use a `for` loop to iterate through the array and add each entry into the `newString` variable.

```
let arr = ['L', 'C', '1', '0', '1'];
let newString = '';

for (i = 0; i < arr.length; i++){
   newString = newString + arr[i];
}

console.log(newString);
console.log(arr);
```

Console Output

```
'LC101'
['L', 'C', '1', '0', '1']
```

Use a `while` loop to add the first element in then remove that element from the array.

```
let arr = ['L', 'C', '1', '0', '
let newString = '';

while (arr.length > 0){
   newString += arr[0];
   arr.shift();
}
console.log(newString);
console.log(arr);
```

Console Output

```
'LC101'
[ ]
```

Inside each loop, the code simply adds two strings together—whatever is stored in `newString` plus one element from the array. In the `for` loop, the element is the next item in the sequence of entries. In the `while` loop, the element is always the first entry from whatever remains in the array.

OK, the loops join the array elements together. Now let’s see how to accomplish the same task without a `for` or `while`statement.

11.7.2. Bring In Recursion Concepts

First, state the problem to solve: *Combine the elements from an array into a string*.

Second, split the problem into small, identical steps: Looking at the loops above, the “identical step” is just adding two strings together - `newString` and the next entry in the array.

Third, build a function to accomplish the small steps: Let’s call the function `combineEntries`, and we will set an array as the parameter.

```
function combineEntries(arrayName){
   //TODO: Add code here
}
```

We want `combineEntries` to repeat over and over again until the task is complete.

How do we make this happen without using `for` or `while`?

11.7.3. Identifying the Base Case

`for` and `while` loops end when a particular condition evaluates to `false`. In the examples above, these conditions are `i <arr.length` and `arr.length > 0`, respectively.

With recursion, we do not know how many times `combineEntries` must be called. To make sure the code stops at the proper time, we need to identify a condition that ends the process. This is called the **base case**, and it represents the simplest possible task for our function.

`if` the base case is `true`, the recursion ends and the task is complete. `if` the base case is `false`, the function calls itself again.

We check for the base case like this:

```
function combineEntries(arrayName){
   if (baseCase is true){
      //solve last small step
      //end recursion
   } else {
      //call combineEntries again
   }
}
```

For the joining task, the *base case* occurs when we pass in a one-element array (e.g. `[ 'L' ]`). With no other elements to join together, the function just needs to return `'L'`.

Let’s update `combineEntries` to check if the array contains only one item.

```
function combineEntries(arrayName){
   if (arrayName.length <= 1){
      return arrayName[0];
   } else {
      //call combineEntries again
   }
}
```

`arrayName.length <= 1` sets up the condition for ending the recursion process. If it is `true`, the single entry gets returned, and the function stops. Otherwise, `combineEntries` gets called again.

Note

We define our base case as `arrayName.length <= 1` rather than `arrayName.length === 1` just in case an empty array `[]` gets passed to the function.

11.7.4. The Case for the Base

What if we accidentally typed `arrayName.length === 2` as the condition for ending the recursion? If so, it evaluates to `true`for the array `['0', '1']`, and the function returns `'0'`. However, this leaves the element `'1'` in the array instead of adding it to the string. By mistyping the condition, we ended the recursion process too soon.

Similarly, if we used `arrayName[0] === 'Rutabaga'` as the condition, then any array that does NOT contain the string `'Rutabaga'` would never match the base case. In situations where the base case cannot be reached, the recusion process either throws an error, or it continues without end—an infinite loop.

Correctly identifying and checking for the base case is *critical* to building a working recursive process.

11.7.5. Check Your Understanding

Question

We can use recursion to remove all of the ‘i’ entries from the array `['One', 'i', 'c', 'X', 'i', 'i',54]`.

Consider the code sample below, which declares the `removeI` function.

```
function removeI(arr) {
   if (baseCase is true){
      //return final array
      //end recursion
   } else {
      //remove one 'i' entry from array
      //call removeI function again
   }
};
```

Which TWO of the following work as a base case for the function? Feel free to test the options in the repl.it to check your thinking.

a. `!arr.includes('i')`
b. `arr.includes('i')`
c. `arr.indexOf('i')===-1`
d. `arr.indexOf('i') !== -1`

Experiment with this repl.it.

Question

The **factorial** of a number (n!) is the product of a positive, whole number and all the positive integers below it.

For example, four factorial is 4! = 4*3*2*1 = 24, and 5! = 5*4*3*2*1 = 120.

Consider the code sample below, which declares the `factorial` function.

```
function factorial(integer) {
   if (baseCase is true){
      //solve last step
      //end recursion
   } else {
      //call factorial function again
   }
};
```

Which of the following should be used as base case for the function?

a. `integer === 1`
b. `integer < 1`
c. `integer === 0`
d. `integer < 0`

Experiment with this repl.it.

11.8. Making A Function Call Itself

Congratulations! Identifying the base case is often the trickiest part of building a recursive function.

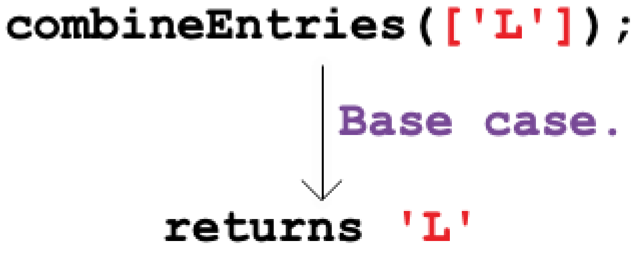
We’ve made it this far with `combineEntries`:

```
function combineEntries(arrayName){
   if (arrayName.length <= 1){
      return arrayName[0];
   } else {
      //call combineEntries again
   }
}
```

Now we are ready to take the next step.

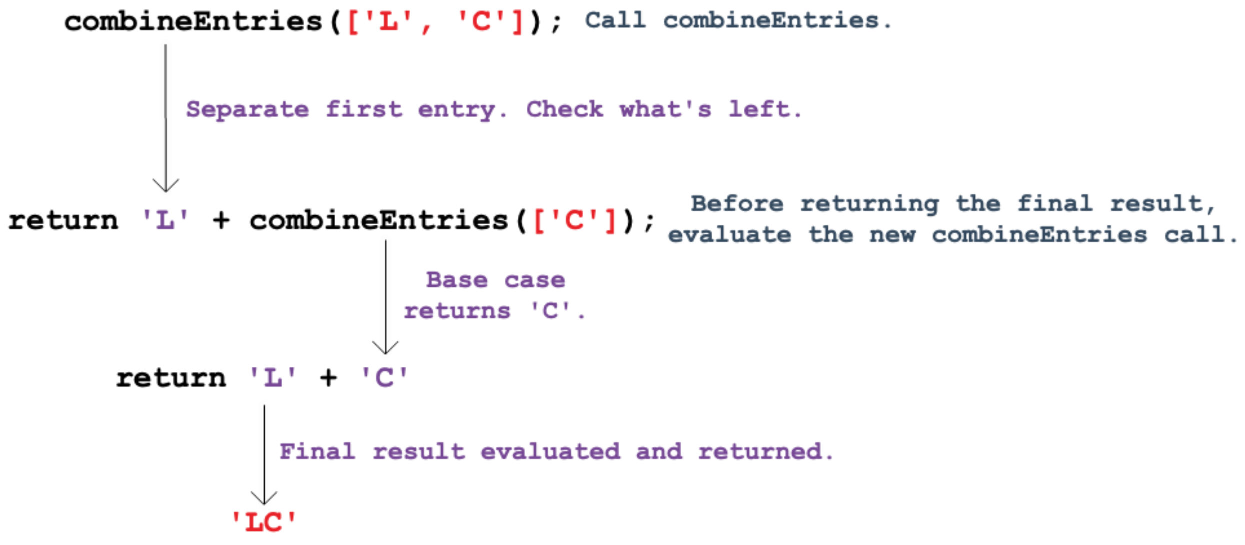
11.8.1. A Visual Representation

To help visualize what happens during recursion, let’s start with the base case `['L']`:



Nothing complicated here. `combineEntries` sees only one item in the array, so it returns `'L'`.

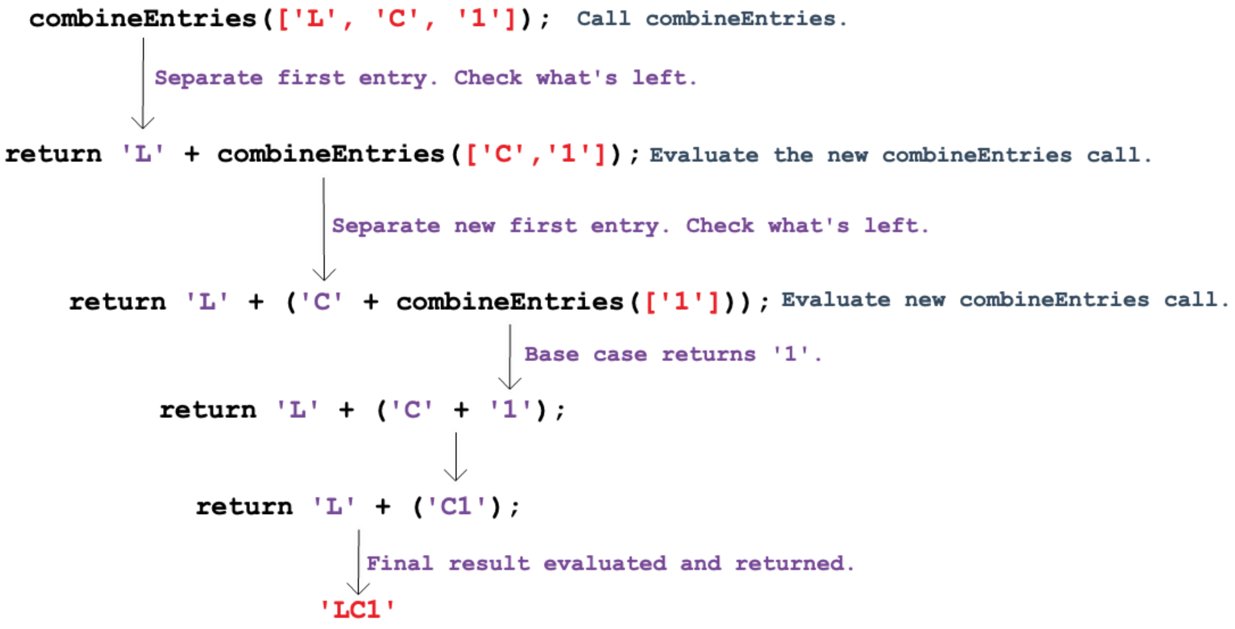
Now consider an array with two elements, `['L', 'C']`:


combineEntries(['L', 'C']); Call combineEntries.
Separate first entry. Check what's left.
return 'L' + combineEntries(['C']);
Before returning the final result, evaluate the new combineEntries call.
Base case returns 'C'.
return 'L' + 'C'
Final result evaluated and returned.
'LC'

In this case, `combineEntries` executes the `else` statement. We have no code for this yet, but we can still consider the logic:

a. `combineEntries` returns `'L'` and calls itself again using what is left inside the array (`['C']`).
b. When passed `['C']`, which is the base case, `combineEntries` returns `'C'`.
c. The strings `'L'` and `'C'` get combined and returned as the final result.

Next, consider an array with three elements `['L', 'C', '1']`:



```
combineEntries(['L', 'C', '1']);  Call combineEntries.
        Separate first entry. Check what's left.
return 'L' + combineEntries(['C','1']); Evaluate the new combineEntries call.
        Separate new first entry. Check what's left.
return 'L' + ('C' + combineEntries(['1'])); Evaluate new combineEntries call.
        Base case returns '1'.
return 'L' + ('C' + '1');
return 'L' + ('C1');
        Final result evaluated and returned.
'LC1'
```

As before, `combineEntries` executes the `else` statement, and we can follow the logic:

a. `combineEntries` returns `'L'` and calls itself again using what is left inside the array (`['C', '1']`).
b. When passed `['C', '1']`, `combineEntries` returns `'C'` and calls itself again using what is left inside the array (`['1']`).
c. When passed `['1']`, which is the base case, `combineEntries` returns `'1'`.
d. The strings `'C'` and `'1'` get combined and returned.
e. The strings `'L'` and `'C1'` get combined and returned as the final result.

As we make the array longer, `combineEntries` calls itself more times. Each call evaluates a smaller and smaller section of the array until reaching the base case. This sets up a series of return events - each one selecting a single entry from the array. Rather than building `'LC101'` from left to right, recursion constructs the string starting with the base case and adding new characters to the front:

Value Returned	Description
`'1'`	Base case. Returns the element from an array of length 1.
`'01'`	Combines the first element from an array of length 2 with the base case value.

Value Returned	Description
`'101'`	Combines the first element from an array of length 3 with the two previous values.
`'C101'`	Combines the first element from an array of length 4 with the three previous values.
`'LC101'`	Combines the first element from an array of length 5 with the four previous values.

Recursive processes all follow this approach. Each call to the function reduces a problem into a slighly smaller piece. The reduction continues until reaching the simplest possible form—the base case. The base case is then solved, and this creates a starting point for completing all of the previous steps.

11.8.2. A Function Calls Itself

So how do we code the `else` statement in `combineEntries`? Recall what needs to happen each time the statement runs:

a. Select the first element in the array,
b. Call `combineEntries` again with a smaller array.

Bracket notation takes care of part a: `arrayName[0]`.

For part b, remember that the slice method returns selected entries from an array. To return everything BUT the first entry in `arr = ['L', 'C', '1', '0', '1']`, use `arr.slice(1)`.

Let's add the bracket notation and the `slice` method to our function:

```
function combineEntries(arrayName){
   if (arrayName.length <= 1){
      return arrayName[0];
   } else {
      return arrayName[0]+combineEntries(arrayName.slice(1));
   }
}
```

Each time the `else` statement runs, it extracts the first element in the array with `arrayName[0]`, then it calls itself with the remaining array elements (`arrayName.slice(1)`).

For `combineEntries(['L', 'C', '1', '0', '1']);`, the sequence would be:

Step	Description
1	First call: Combine `'L'` with `combineEntries(['C', '1', '0', '1'])`.
2	Second call: Combine `'C'`, with `combineEntries(['1', '0', '1'])`.

Step	Description
3	Third call: Combine `'1'`, with `combineEntries(['0', '1'])`.
4	Fourth call: Combine `'0'`, with `combineEntries(['1'])`.
5	Fifth call: Base case returns `'1'`.

To get the final result, proceed *up the chain*:

Step	Description
5	Return `'1'` to the fourth call.
4	Return `'01'` to the third call.
3	Return `'101'` to the second call.
2	Return `'C101'` to the first call.
1	Return `'LC101` as the final result.

See this recursion in action.

11.8.3. Check Your Understanding

Question

What if we wanted to take a number (n) and add it to all of the positive integers below it? For example, if n = 5, the function returns 5 + 4 + 3 + 2 + 1 = 15.

Consider the code sample below, which declares the `decreasingSum` function.

```
function decreasingSum(integer) {
   if (integer === 1){
      return integer;
   } else {
      //call decreasingSum function again
   }
}
```

Which of the following should be used in the `else` statement to recursively call `decreasingSum` and eventually return the correct answer?

a. `return integer + (integer-1);`
b. `return integer + (decreasingSum(integer));`
c. `return integer + (decreasingSum(integer-1));`
d. `return decreasingSum(integer-1);`

Experiment with this repl.it.

11.9. Recursion Wrap-Up

In order to function (ba-dum chhhh), recursion must fulfill four conditions:

1. A series of small, identical steps combine to solve a larger problem.
2. A base case must be defined. When true, this simplest case halts the recursion.
3. A recursive function repeatedly calls itself.
4. Each time the recursive function is called, it must alter the data/variables/conditions in order to move closer to the base case.

Benefits of Recursion:

a. Fewer lines of code required to accomplish a task,
b. Makes code cleaner and more readable.

Drawbacks of Recursion:

a. More abstract than using loops,
b. Code is “more readable” only if the reader understands recursion.

11.9.1. Recursion in a Nutshell

a. Build a single function to break a big problem into a slightly smaller version of the *exact same problem*.
b. The function repeatedly calls itself to reduce the problem into smaller and smaller pieces.
c. Eventually, the function reaches a simplest case (the *base*), which it solves.
d. Solving the base case sets up the solutions to all of the previous steps.

11.9.2. Why Do I Need To Know Recursion?

If you ask veteran programmers how often they use recursion, you will get answers ranging from “Not since I had to do it in school,” to “Very regularly.” Some programmers avoid recursion like the plague, while others look forward to using it wherever it fits.

Most of the recursion problems you encounter in your tech career can be solved with loops instead. However, *recursion is a skill most programmers will see and are expected to know*, even if they do not use it all the time. How deep you need to dive depends entirely on the type of job you get, your team members, and your personal preference.

Let’s use an analogy. At some point in time, most teens must “solve a quadratic” in school (e.g. find ‘x’ in $x^2 + 2x - 35 = 0$). Perhaps you fondly remember doing this yourself. As kids, we were *expected* to know how to solve a quadratic, but as adults, the need to do this varies. Some of us must frequently find x, while others only need to solve one or two equations a year. Still others do not see quadratics again until their own kids learn about them.

Since their future jobs might not require it, why do teens need to learn how to solve quadratics? Because at some point in time they will have to do it again (if only to shock their kids), and they need to be ready when that happens.

The same is true for recursion.

Learn it. Love it. Use it.

11.10. Exercises: More Functions

11.10.1. Practice Yer Skills

Arrr! Welcome back, swabbie. Ye be almost ready fer yer next task—earning gold fer yer cap'n.

First, ye need to show what ye've learned.

11.10.1.1. Complete the Map

Not THAT kinda map, ye rat! Fold that up and do the following:

1. Create an anonymous function and set it equal to a variable. Yer function should:
   a. If passed a number, return the tripled value.
   b. If passed a string, return the string "ARRR!"
   c. If NOT passed a number or string, return the data unchanged.
   d. Build yer function here, and be sure to test it.
2. Add to yer code! Use yer function and the map method to change the array `['Elocution', 21, 'Clean teeth',100]` as follows:
   a. Triple all the numbers.
   b. Replace the strings with "ARRR!"
   c. Print the new array to confirm yer work.

11.10.2. Raid Yonder Shuttle

Shiver me timbers! Ye did well. Yer ready ta join a boarding party.

Ye may still be wonderin' *Why the blazes would I ever use an anonymous function?* Fer today's mission, o' course! We arrrr going to loot yonder shuttle, but since it's the 21st century, we need to do better than grappling hooks and rope.

Ye arrrr going to hack into the shuttle code and steal supplies. Since the LaunchCode TAs keep a sharp eye on the goodies, ye can't just add new functions like `siphonFuel` or `lootCargo`. Ye need to be more sneaky.

Clever.

Invisible.

Anonymous.

The first mate swiped a copy of the code ye need ta hack:

```
function checkFuel(level) {
  if (level > 100000){
    return 'green';
  } else if (level > 50000){
    return 'yellow';
  } else {
    return 'red';
  }
}

function holdStatus(arr){
  if (arr.length < 7) {
     return `Spaces available: ${7 - arr.length}`;
  } else if (arr.length > 7){
     return `Over capacity by ${arr.length - 7} items.`
  } else {
     return "Full";
  }
}

let fuelLevel = 200000;
let cargoHold = ['meal kits', 'space suits', 'first-aid kit', 'satellite', 'gold', 'water', 'AI

console.log("Fuel level: "+ checkFuel(fuelLevel));
console.log("Hold status: "+ holdStatus(cargoHold));
```

Hack the code at repl.it.

1. First, steal some fuel from the shuttle:
   a. Define an anonymous function and set it equal ter a variable with a normal, non-suspicious name. The function needs one parameter, which will be the fuel level on the shuttle.
   b. Ye must siphon off fuel without alerting the TAs. Inside yer function, ye want to reduce the fuel level as much as possible WITHOUT changing the color returned by the `checkFuel` function.
   c. Once ye figure out how much fuel ter pump out, return that value.
   d. Decide where to best place yer function call to gather our new fuel.
   e. Be sure to test yer function! Those bilge rat TAs will notice if they lose too much fuel.
2. Next, liberate some of that glorious cargo.
   a. Define another anonymous function with an array as a parametarrrrr, and set it equal to another innocent variable.
   b. Ye need to swipe two items from the cargo hold. Choose well. Stealing water ain't gonna get us rich. Put the swag into a new array and return it from the function.
   c. The cargo hold has better security than the fuel tanks. It counts how many things are in storage. Ye need to replace what ye steal with something worthless. The count MUST stay the same, or ye'll get caught and thrown into the LaunchCode brig.
   d. Don't get hasty, swabbie! Remember to test yer function.
3. Finally, ye need to print a receipt for the accountant. Don't laugh! That genius knows MATH and saves us more gold than ye can imagine.
   a. Define a function called `irs` that takes `fuelLevel` and `cargoHold` as parametarrrrrs.
   b. Call yer anonymous fuel and cargo functions from within `irs`.
   c. Use a template literal to return, `"Raided _____ kg of fuel from the tanks, and stole ____ and ____ fromthe cargo hold."`

11.11. Studio: More Functions

11.11.1. Sort Numbers For Real

Recall that using the `sort` method on an array of numbers produced an unexpected result, since JavaScript converts the numbers to strings by default. Let's fix this!

Here is one approach to sorting an array:

1. Find the minimum value in an array,
2. Add that value to a new array,

3. Remove the entry from the old array,
4. Repeat steps 1 - 3 until the numbers are all in order.

11.11.1.1. Part A: Find the Minimum Value

Create a function with an array of numbers as its parameter. The function should iterate through the array and return the minimum value from the array.

Hint: Use what you know about `if` statements to identify and store the smallest value within the array.

Tip

Use this sample data for testing.

```
[ 5, 10, 2, 42 ]
[ -2, 0, -10, -44, 5, 3, 0, 3 ]
[ 200, 5, 4, 10, 8, 5, -3.3, 4.4, 0 ]
```

Code studio part A at repl.it.

11.11.1.2. Part B: Create a New Sorted Array

Create another function with an array of numbers as its parameter. Within this function:

a. Define a new, empty array to hold the final sorted numbers.
b. Use your function from the previous exercise to find the minimum value in the old array.
c. Add the minimum value to the new array, and remove the minimum value from the old array.
d. Repeat parts b & c until the old array is empty.
e. Return the new sorted array.
f. *Be sure to print the results in order to verify your code*.

Code studio part B at repl.it.

Tip

Which type of loop?

Either a `for` or `while` loop will work inside this function, but one IS a better choice. Consider what the function must accomplish vs. the behavior of each type of loop. Which one best serves if the array has an unknown length?

11.11.2. More on Sorting Numbers

The sorting approach used above is an example of a *selection sort*. The function repeatedly checks an array for the minimum value, then places that value into a new container.

Selection sorting is NOT the most efficient way to accomplish the task, since it requires the function to pass through the array once for each item within the array. This takes way too much time for large arrays.

Fortunately, JavaScript has an elegant way to properly sort numbers.

Tip

Here is a nice, visual comparison of different sorting methods.

Feel free to Google “bubble sort JavaScript” to explore a different way to order numbers in an array.

11.11.3. Part C: Number Sorting the Easy Way

If you Google “JavaScript sort array of numbers” (or something similar), many options appear, and they all give pretty much the same result. The sites just differ in how much detail they provide when explaining the solution.

One reference is here: W3Schools.

End result: the JavaScript syntax for numerical sorting is `arrayName.sort(function(a, b){return a-b});`.

Here, the anonymous function determines which element is larger and swaps the positions if necessary. This is all that `sort`needs to order the entire array.

Using the syntax listed above:

a. Sort each sample array in increasing order.
b. Sort each sample array in decreasing order.
c. Does the function alter `arrayName`?
d. Did your sorting function from part B alter `arrayName`?

Code studio part C at repl.it.

11.11.4. So Why Write A Sorting Function?

Each programming language (Python, Java, C#, JavaScript, etc.) has built-in sorting methods, so why did we ask you to build one?

It’s kind of a programming right of passage - design an efficient sorting function. Also, sorting can help you land a job.

As part of a tech interview, you will probably be asked to do some live-coding. One standard, go-to question is to sort an array WITHOUT relying on the built in methods. Knowing how to think though a sorting task, generate the code and then clearly explain your approach will significantly boost your appeal to an employer.

11.11.5. Bonus Mission

Refactor your sorting function from Part B to use recursion.