

Arrow Functions (Fat Arrows)

Arrow functions create a concise expression that encapsulates a small piece of functionality. Additionally, arrows retain the scope of the caller inside the function eliminating the need of self = this.

Example

```
// const multiply = function(x,y) {
//   return x * y;
// }
// Can be rewritten as:
// const multiply = (x, y) => { return x * y };
// Since the function is a single expression return and braces are not needed.
const multiply = (x, y) => x * y;
console.log(multiply(5,10)) //50
```

Function Delegates

Function delegates encapsulate a method allowing functions to be composed or passed as data.

Example

```
const isZero = n => n === 0;
const a = [0,1,0,3,4,0];
console.log(a.filter(isZero).length); // 3
```

Expressions Instead of Statements

Statements define an action and are executed for their side effect. Expressions produce a result without mutating state.

Statement

```
const getSalutation = function(hour) {
  var salutation; // temp value
  if (hour < 12) {
    salutation = "Good Morning";
  }
  else {
    salutation = "Good Afternoon"
  }
  return salutation; // mutated value
}
```

Expression

```
const getSalutation = (hour) => hour < 12 ? "Good Morning" : "Good Afternoon";
console.log(getSalutation(10)); // Good Morning'
```

Higher Order Functions

A function that accepts another function as a parameter, or returns another function.

Example

```
function mapConsecutive(values, fn) {
  let result = [];
  for(let i=0; i < values.length -1; i++) {
    result.push(fn(values[i], values[i+1]));
  }
}
```



Higher Order Functions (cont)

```
}
  return result;
}
const letters = ['a','b','c','d','e','f','g'];
let twoByTwo = mapConsecutive(letters, (x,y) => [x,y]);
console.log(twoByTwo);
// [[a,b], [b,c], [c,d], [d,e], [e,f], [f,g]]
```

Method Chaining

Method chains allow a series of functions to operate in succession to reach a final result. Method chains allow function composition similar to a pipeline.

Example

```
let cart = [{name: "Drink", price: 3.12},
  {name: "Steak", price: 45.15},
  {name: "Drink", price: 11.01}];
let drinkTotal = cart.filter(x=> x.name === "Drink")
  .map(x=> x.price)
  .reduce((t,v) => t +=v)
  .toFixed(2);
console.log(Total Drink Cost $${drinkTotal}); // Total Drink Cost $14.13
```

Currying

Currying allows a function with multiple arguments to be translated into a sequence of functions. Curried functions can be tailored to match the signature of another function.

Example

```
const convertUnits = (toUnit, factor, offset = 0) => input => ((offset + input) * factor).toFixed(2).concat(toUnit);
const milesToKm = convertUnits('km', 1.60936, 0);
const poundsToKg = convertUnits('kg', 0.45460, 0);
const fahrenheitToCelsius = convertUnits('degrees C', 0.5556, -32);
milesToKm(10); // "16.09 km"
poundsToKg(2.5); // "1.14 kg"
fahrenheitToCelsius(98); // "36.67 degrees C"
const weightsInPounds = [5,15.4,9.8, 110];
// const weightsInKg = weightsInPounds.map(x => convertUnits('kg', 0.45460,
0)(x));
// with currying
const weightsInKg = weightsInPounds.map(poundsToKg);
// 2.27kg, 7.00kg, 4.46kg, 50.01kg
```



By **EllieAtWHL** (EllieAtWHL)
cheatography.com/elliematwhl/

Published 6th March, 2020.
Last updated 7th April, 2021.
Page 2 of 3.

Sponsored by **Readable.com**
Measure your website readability!
<https://readable.com>

Array Manipulation Functions

Array Functions are the gateway to functional programming in JavaScript. These functions make short work of most imperative programming routines that work on arrays and collections.

```
[].every(fn)
```

Checks if all elements in an array pass a test.

```
[].some(fn)
```

Checks if any of the elements in an array pass a test.

```
[].find(fn)
```

Returns the value of the first element in the array that passes a test.

```
[].filter(fn)
```

Creates an array filled with only the array elements that pass a test.

```
[].map(fn)
```

Creates a new array with the results of a function applied to every element in the array.

```
[].reduce(fn(accumulator, currentValue))
```

Executes a provided function for each value of the array (from left-to-right). Returns a single value, the accumulator.

```
[].sort(fn(a,b)) warning, mutates state!
```

Modifies an array by sorting the items within an array. An optional compare function can be used to customize sort behaviour. Use the spread operator to avoid mutation. [...arr].sort()

```
[].reverse() warning, mutates state!
```

Reverses the order of the elements in an array. Use the spread operator to avoid mutation. [...arr].reverse()

Pipelines

A pipeline allows for easy function composition when performing multiple operations on a variable. Since JavaScript lacks a Pipeline operator, a design pattern can be used to accomplish the task.

Example

```
const pipe = functions => data => {
  return functions.reduce(
    (value, func) => func(value),
    data
  );
};

let cart = [3.12, 45.15, 11.01];
const addSalesTax = (total, taxRate) => (total * taxRate) + total;
const tally = orders => pipe([
  x => x.reduce((total, val) => total + val), // sum the order
  x => addSalesTax(x, 0.09),
  x => Order Total = ${x.toFixed(2)} // convert to text
])(orders); // Order Total = 64.62
```



By **EllieAtWHL** (EllieAtWHL)
cheatography.com/elliematwhl/

Published 6th March, 2020.
 Last updated 7th April, 2021.
 Page 3 of 3.

Sponsored by **Readable.com**
 Measure your website readability!
<https://readable.com>