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 = functi on(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;
consol e.l og( mul tip ly( 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];
consol e.l og( a.f ilt er( isZ ero ).l ength); // 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 getSal utation = functi on( hour) {
 var saluta tion; // temp value
 if (hour < 12) {
    salutation = "Good Mornin g";
}
 else {
    salutation = "Good Aftern oon "
}
  return saluta tion; // mutated value
}
```

Expression

```
const getSal utation = (hour) => hour < 12 ? "Good Mornin g" : "Good Aftern oon ";
consol e.l og( get Sal uta tio n(10)); // Good Morning'
```

Higher Order Functions

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

Example

```
function mapCon sec uti ve( values, fn) {
 let result = [];
  for(let i=0; i < values.length -1; i++) {
    res ult.pu sh( fn( val ues[i], values [i+ 1]));
```


Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com

Higher Order Functions (cont)

```
 }
  return result;
}
const letters = ['a',' b', 'c' ,'d ',' e', 'f' ,'g'];
let twoByTwo = mapCon sec uti ve( let ters, (x,y) => [x,y]);
consol e.l og( two ByTwo);
// [[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: " Dri nk", price: 3.12},
  {name: " Ste ak", price: 45.15},
  {name: " Dri nk", price: 11.01}];
let drinkTotal = cart.f ilt er( x=> x.name === " Dri nk")
 .ma p(x => x.price)
 .re duc e((t,v) => t +=v)
 .to Fix ed(2);
consol e.l og( Total Drink Cost $${dri nkT otal}); // 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 conver tUnits = (toUnit, factor, offset = 0) => input => ((offset + input) * factor ).t oFi xed (2).
co nca t(t oUnit);
const milesToKm = conver tUn its ('km', 1.60936, 0);
const poundsToKg = conver tUn its ('kg', 0.45460, 0);
const farenh eit ToC elsius = conver tUn its ('d egrees C', 0.5556, -32);
milesT oKm (10); //"16.09 km"
pounds ToK g(2.5); //"1.14 kg"
farenh eit ToC els ius (98); //"36.67 degrees C"
const weight sIn Pounds = [5,15.4 ,9.8, 110];
// const weight sInKg = weight sIn Pou nds.map(x => conver tUn its ('kg', 0.45460,
0)(x));
// with currying
const weight sInKg = weight sIn Pou nds.ma p(p oun dsT oKg);
// 2.27kg, 7.00kg, 4.46kg, 50.01kg
```


Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.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.

`[].eve ry(fn)`

Checks if all elements in an array pass a test.

`[].som e(fn)`

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

`[].fin d(fn)`

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

`[].fil ter(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.

`[].red uce (fn (ac cum ulator, curren tVa lue))`

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

`[].sor t(f n(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. `[...ar r].s ort()`

`[].rev erse()` *warning, mutates state!*

Reverses the order of the elements in an array. Use the spread operator to avoid mutation. `[...ar r].r ev erse()`

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 functi ons.re duce(
    (value, func) => func(v alue),
 data
 );
};
let cart = [3.12, 45.15, 11.01];
const addSal esTax = (total, taxRate) => (total * taxRate) + total;
const tally = orders => pipe([
x => x.redu ce( (total, val) => total + val), // sum the order
x => addSal esT ax(x, 0.09),
x => Order Total = ${x.to Fix ed(2)} // convert to text
])(ord ers); // Order Total = 64.62
```


Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com