Arrow Functions

Understanding and utilizing anonymous functions is important to becoming a skilled JavaScript developer. An **anonymous function** is simply a function *without a name*. ES6 introduces a new shortened syntax for writing anonymous functions that is the focus of this lesson.

Function Keyword

Consider a simple sayHello function assigned to a variable.

*var* sayHello = function(name) {

console.log('Hello ' + name);

};

copy

Utilizing ES6 arrow functions, colloquially fat arrow functions, we can rewrite it as such:

*const* sayHello = (name) *=>* {

console.log(`Hello ${name}`);

};

copy

Notice we've omitted the function keyword and now have an arrow (=>) pointing to the function body. Interesting, but what benefits does this provide other than less typing?

For simple methods we can refine this example further. Single parameters don't need parenthesis and with the function body being a single statement we can remove the curly braces.

*const* sayHello = name *=>* console.log(`Hello ${name}`);

copy

Concise. More complex functions will need a more complete body ({}), and multiple parameters will require parenthesis. Another benefit of utilizing arrow functions for simple expressions is implicit returns. Let's create a new example.

*var* square = function(n) {

return n \* n;

};

copy

will become

*const* square = n *=>* n \* n;

copy

Traditional functions require explicit returns, such as in the first square. With arrow functions the result of our expressions, n \* n, is implicitly returned to the caller.

Arrow functions give us the ability to shorten the syntax of returning an object as well. However, there is one piece of syntax we need in order to return an object. Let's look at the example below.

// longhand notation to return an object

// NOTE: first set of brackets are defining the function body

// and the second set of brackets are to create the object literal

*const* returnObjLonghand = () *=>* {

    return {

        firstName: 'John',

        lastName: 'Wick'

    }

}

/\*\*

\* The example below wouldn't work because the

\* brackets are interpreted as opening the body of the

\* function rather than brackets to create an object literal

\*/

*const* returnObj = () *=>* { firstName: 'John', lastName: 'Wick' }

// surrounding the implicit return with parenthesis solves the problem

*const* returnObjFixed = () *=>* ({ firstName: 'John', lastName: 'Wick' });copy

Context

Fancy syntax isn't the only change with arrow functions. They also inherit context from the parent scope. To demonstrate this let's create a card Deck class.

*class* Deck {

constructor() {

*const* suits = ['Diamond', 'Heart', 'Spade', 'Club'];

*const* faces = ['Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King'];

*const* deck = [];

for (*const* suit of suits) {

for (*const* face of faces) {

deck.push(this.createCard(suit, face));

}

}

this.deck = deck;

}

}

copy

We'll start refactoring for a more functional approach using forEach, but continue using traditional anonymous functions.

*class* Deck {

  constructor() {

*const* suits = ['Diamond', 'Heart', 'Spade', 'Club'];

*const* faces = ['Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King'];

*const* deck = [];

     suits.forEach(function(suit) {

    faces.forEach(function(face) {

  deck.push(this.createCard(suit, face));

     });

  });

     this.deck = deck;

    }

}

copy

Assume we have a createCard method in our class. Creating a new Deck will now result in a TypeError: TypeError: Cannot read property 'createCard' of undefined, because this in our anonymous functions doesn't have the same context as our loops.

We can fix this issue with arrow functions like so:

*class* Deck {

constructor() {

...

suits.forEach(suit *=>* {

faces.forEach(face *=>* {

deck.push(this.createCard(suit, face));

});

});

}

}

copy

Arrow functions don't create their own context, they look to their enclosing scope for that information. Therefore, this should now refer to the Deck instance, which has a createCard method

Conclusion

Arrow functions provide a simpler syntax with implicit returns for succinct expressions. Additionally, they inherit context from parent scopes. While this is great much of the time, there are instances when you want context to change, so be aware of how an arrow function might affect your code.