Arrow Functions

Two benefits to arrow functions

1. Less verbose than traditional function expressions
   1. arr.map(x=>x\*x) vs. arr.map(function (x) { return x\*x });
2. their this is picked up from surroundings (lexical) - > you no longer need bind or that = this
   1. following variables are all lexical inside arrow functions – their values are determined by surrounding scope
      1. arguments
      2. super
      3. this
      4. new.target
   2. traditional functions have a dynamic this, its value is determined by how they are called

Traditional functions are bad non-method functions because of functions’ own this

Traditional functions can be used as

1. non-method functions
2. methods
3. constructors

because of 2 and 3, functions always have their own this

this prevents accessing *this* of a surrounding method from inside a callback

what you are left with is having to add extra code to perform work arounds for the problems that the code introduces

the fat arrow => was chosen to be compatible with CoffeeScript

in fact it operates the same way

two types of blocks for functions are now possible

statement block behaves like a normal function body

and now we can use expression blocks which are just like expressions in statements and imply the same thing – that the expression is always implicitly returned

* with this it is important to remember that you do not write x=>return x; that will actually return an error

omitting the parentheses around the parameters is only possible if they consist of a single identifier

as soon as there is anything else, including destructuring of a single parameter or providing a default value for a single param, you need to type the parentheses

=> can be viewed as an operator that binds loosely, other operators that it could potentially be in conflict with will win

This is important so that expressions on the right side of an arrow function can ‘stick together’

In this function

const f = x => (x %2) === 0 ? x : 0;

we want => to evaluate last, and the others to evaluate first, so loose binding is important

as a consequence you often have to wrap arrow functiosn in parentheses if they compete with other operators

console.log(typeof () => {}) produces an error

console.log(typeof (()=> {})) is okay

this forces the stuff inside the parentheses to be evaluated completely first

you must put parameters for an arrow function on the same line as the arrow

if you use a statement with an arrow function you have to wrap it in curlies

if you want to return an object from an arrow, you have to wrap the object in parentheses – this is to make it clear that the object is in fact not a function block

IIFE = Immediately Invoked Function Expression is an ES5 construct (function() {return 123;})();

IIAF = Immediately Invoked Arrow Function (() => {return 123;})();

For consistency with arrow functions whose bodies are expressions, you must wrap an IIAF in parentheses for IIAFs even if they have a function body

Because there is loose binding with arrow functions, parentheses that follow an expression body of an arrow function would evaluate to be part of the expression body, if no parentheses wrapped the arrow function

const value = () => foo()

evaluates to

const value = () => (foo());

NOT

const value = (()=>foo)();

arrow functions can’t be used as a constructor

they do not have an internal method Construct and they do not have the property prototype

there are no other observable differences between the two

Chapter 12

As callbacks, arrow functions have two advantages over traditional functions:

• this is lexical and therefore safer to use.

• Their syntax is more compact. That matters especially in functional programming, where there are many higher-order functions and methods (functions and methods whose parameters are functions).

• Arrow functions are made for non-method functions. They pick up this from their surrounding scopes (“lexical this”).

• Method definitions are made for methods. They provide support for super, to refer to super-properties and to make super-method calls.

As stand-alone functions (versus callbacks), I prefer function declarations:

function foo(arg1, arg2) { ··· }

The benefits are:

• Subjectively, I find they look nicer. In this case, the verbose keyword function is an advantage – you want the construct to stand out. • They look like generator function declarations, leading to more visual consistency of the code.

There is one caveat: Normally, you don’t need this in stand-alone functions. If you use it, you want to access the this of the surrounding scope (e.g. a method which contains the stand-alone function). Alas, function declarations don’t let you do that – they have their own this, which shadows the this of the surrounding scope. Therefore, you may want to let a linter warn you about this in function declarations. Another option for stand-alone functions is assigning arrow functions to variables. Problems with this are avoided, because it is lexical.

const foo = (arg1, arg2) => { ··· };

With regard to names, arrow functions are like anonymous function expressions:

const func = () => {}; console.log(func.name); // func

From now on, whenever you see an anonymous function expression, you can assume that an arrow function works the same way.