81)Arrow Functions: Part2

Here we will see difference between arrow function syntax and normal function. in case of regular functions we can use arguments array inside function body , like this-

const add = function(a,b) {

return arguments[0] + arguments[1];

}

console.log(add(2, 8));

but in case of arrow functions , we cannot.

const add = (a,b) => {

return arguments[0] + arguments[1];

}

console.log(add(2, 8));

we get error which is specific to nodejs. Error is-

**[object Object]function require(path) {**

**try {**

**exports.requireDepth += 1;**

**return mod.require(path);**

**} finally {**

**exports.requireDepth -= 1;**

**}**

**}**

But we don’t use argument inside function body most of time.so its not a downside.

Another difference is that arrow functions don’t bind their this value. meaning they are bad candidates for most methods. Ex, this code works fine-

const car = {

color: 'Red',

getSummer: function() {

return `The car is ${this.color}`;

}

};

console.log(car.getSummer());

however if we use arrow function-

const car = {

color: 'Red',

getSummer: () => {

return `The car is ${this.color}`;

}

};

console.log(car.getSummer());

output-

**The car is undefined**

So it is trying to access something that does not exist. So it means when we are creating methods(functions as object properties) we whould stick to regular function as opposed to arrow functions.

Now there is still an alternative way to define your regular functions on an object. this is using method definition syntax. Here is syntax-

const car = {

color: 'Red',

getSummer () {

return `The car is ${this.color}`;

}

};

This is still regular old function that does bind this. This is just alternative syntax for setting up a method. So these are diffrences- arrow function don’t bind arguments, they don’t bind this.

To study mote visit these links-

<https://medium.com/@thejasonfile/es5-functions-vs-es6-fat-arrow-functions-864033baa1a>

<https://dmitripavlutin.com/when-not-to-use-arrow-functions-in-javascript/>

82)Condition(Ternary )Operator

const age = 7;

const message = age >= 18 ? 'You can Vote' : 'You cannot Vote';

console.log(message);

const myAge = 27;

const showPage = () => {

console.log('Showing the page');

};

const showErrorPage = () => {

console.log('Showing the error page');

};

myAge >= 21 ? showPage() : showErrorPage;

here we used ternary operator to assign value and call function.

83)Truth and falsy value

Undefined is treated as falsy value. see below code-

const products = [];

const product = products[0];

if (product !== undefined) {

console.log('Product Found');

} else {

console.log('Product not found');

}

What I want to talk about is what we put inside of if condition, we put Boolean or some expression that resolve to a Boolean. There is another way we can write this if condition. this is using truthy and falsy.

Every single value in javascript, anything you can possible imagine is going to either be truthy or falsy. This means when evaluated in Boolean context that value either ends up being true or false.

So truthy means value that resolve o true in Boolean context. boolean context means, place in code where we expect Boolean, like in if statement. Opposite is truthy is falsy.

If we do this-

if ('testing') {

}

Our program won’t crash, javascript will do its best to convert this non Boolean value into Boolean. That sis where truthy and falsy definations comes into place. It’s easier to define what is falsy and then we know that everything else is truthy. Following are falsy values-

1)0

2)empty string

3)null

4)undefined

5)NaN

Everything else including empty array , empty object({}) is true.

So truthy values cn help us to clean our code. We can use them in if condition and ternary operator. Ex-

const products = [];

const product = products[0];

if (product) {

console.log('Product Found');

} else {

console.log('Product not found');

}

if ('testing') {

}

We can also filp truthy or falsy values with not operator.

let name;

if (!name) {

console.log('This staement will be executed');

}

84)Type Coercion

In last lecture we studied about truthy and falsy values. This is actually part of bigger language feature called type coercion. This is allowing us to take a value of one type and in a specific context automatically translate it over to value of different type. In last lecture we coerced values to boolean value. In this case we wanted to do type coercion, but there are lot of of other situations where we are going to get very strange and unexpected behaviour.

console.log('5' + 5);

as a output we get 55.

So this is javascript not being 100% sure what it should be doing. So it takes its best guess at what behaviour you want and now type coercion in this fashion is typically something we want to avoid because we are going to get extremely inconsistent behaviour. Ex-

console.log('5' - 5);

output – 0;

difference between 2 statements is that we replaced + sign with-. Now once gain we hve to guess in which context is this going to operate. Is it going to operate in string context or it is going to operate in number context. This cannot operate in string context because we cannot minus 2 strings. So this operate under context of number.

So this is inconsistent behaviour I am talking about. We should avoid this situration where we are trying to perform operations on values of 2 different types. This is type coercion and as we saw it can be very useful when trying to convert things over to Boolean, buth when it comes to other types we want to avoid it. We want to avoid it with strings and with numbers. There are 3 values you can coerce to- number, string and Boolean. That’s it. We saw and string and number version above and we saw Boolean version in last lecture. W are going to avoid number and string versions at all costs but coercion to Boolean is super common and its ok.

There I another type of type coercion that I want to cover. I am covering just for sake of completeness. we want to avoid it.

console.log('5' == 5);

here we get true. We also have === operator. It checks to value as well as type equaity.

To check type we have **typeof** operator. This operator returns a string. Code-

const value = true + 12;

console.log(typeof value);

console.log(value);

output – number, 13

here Boolean is converted into number. True is converted into 1 and false into 0. So that’s it. If we want to dive more then there is a link to a article on type coercion.

85)Catching and throwing Errors

Here we will learn how to handle error in js code.lets say we have function that accepts number as a argument and we perform some math operation on it. It will work fine when we pass number to it. But when someone passes some argument other than number, we want to let them know they are misusing the number.

If someone passes in number we will get number. But what if someone misuses the function passing in something else, something that gives us an unexpected but potentially valid result. Ex-

const getTip = (amount) => {

return amount \*25;

};

const resut = getTip(true);

console.log(result);

here we pass Boolean instead of number, boolean is converted into number 1 and our program won’t crash. This is not what we want. So we do this-

const getTip = (amount) => {

if(typeof amount === 'number') {

return amount\*25;

} else {

throw 'Argument must bea number';

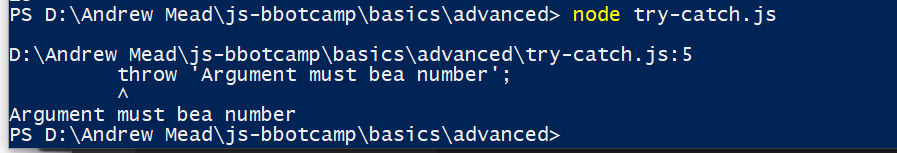
}

};

const resut = getTip(true);

console.log(result);

output-



Here in else block we are throwing a error. When we throw an error it crashes our program. We are allowed to provide a message as well as bit of context as to where the problem was. In this case t is in this file on some line. So here we throw a custom error , so that caller knows that he has misused the function. to do this we use throw statement. After throw, we provide some more information about error. Now this could be anything and could be boolen,number or any other type, but in almost all situations it is string that describes exactly what went wrong.

When we run the program we can see that our program crashed and we can see the error on console.

But if you remember, in error we have way more than just couple of lines related to error. When we throw string we just get string back at console. There is a built in function that we can use to generate more comprehensive error. So we use throw function.

const getTip = (amount) => {

if(typeof amount === 'number') {

return amount\*25;

} else {

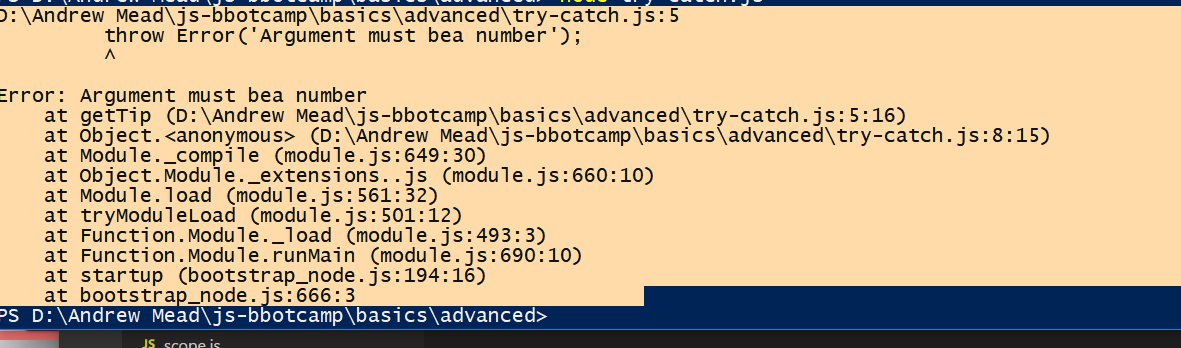
throw Error('Argument must bea number');

}

};

const resut = getTip(true);

console.log(result);



first 4 lines are same but stuff down is different. This stack trace shows the various pieces of code running to get to the point in application where the error was thrown. First it showns at which line of function is error. Then it tells at what line was function called.This can be useful if we are calling this function 15 times. So by throwing a real error, we are getting a bit more context about why and where that one came from. So we will use this second way to throw error.

Now just because program throws an error does’nt mean that we have to let our program crash and burn , like we are already doing. Not all errors require code change to be fixed. To do this we use try-catch block. In try block we write some code, if there is no error then it is fine, we will continue with rest of program. But error is thrown, then control comes to catch block. Code-

const getTip = (amount) => {

if(typeof amount === 'number') {

return amount\*25;

} else {

throw Error('Argument must bea number');

}

};

try {

const resut = getTip(true);

console.log(result);

console.log('catch block is running');

}

Output-

**catch block is running**

Now lets say our function is from 3rd party library. we do not know about its implementation . we call function in try block. If everything is fine, try block is executed and then rest of the program. But in case of error, catch block is executed.When we throw a error in try block. Then lines in catch block are executed.

We passed a argument e to catch block. We can access the string passed in error with this. Just do e.message.

86)Handling application errors

Here we introduced try catch block in our code. We read data from local storage parse it and return array. But our application will crash, if data in local storage is not json. So we used try catch to handle this block.

const getSavedNotes = () => {

const notesJSON = localStorage.getItem('notes');

try {

return notesJSON ? JSON.parse(notesJSON) : [];

} catch(e) {

return [];

}

};

87)Working in strict Mode

Here we will learn about strict mode in javascript. Strict mode is different mode in which we can run our js. and when we opt in to this mode we are opting into slightly better version of the language. Some of easy to fall into traps and weird language quirks are tweaked making it lot harder to mess up. Code-

const processData = () => {

data = 27;

};

processData();

console.log(data);

here we are trying to write to undeclared variable. So js is going to create data variable in global scope. But if we enable strict mode, then we get error-

'use strict'

const processData = () => {

data = 27;

};

processData();

console.log(data);

that error tells us that data variable is not defined.

Another use of strict mode is that it makes our code future proof. Think about reserved keywords in language. We have thinks like let and const, these can’t be used for variable names for example. Well there are other things that are coming to javascript in future but they are not quite here yet . with strict mode though we can reserve those identifiers. Making sure they are not used in program. Ex-

We can made variable with name public . while in strict mode we cannot. Public is reserved keyword for future.

Let use strict in all js files of notes and todo app.

There are more features to strict for example there are parts of language that were popular in the past, that have been deprecated. Meaning they are not removed from language but they might be removed in future. So they are not recommended. Use strict disables those.

To read more about strict mode- link to article is given in pdf guide.