**Report about**

“Differences Between Arrow and Regular Functions”

React Training – Task #5

The report includes what are the functions of the Arrow and the Regular and the difference between them, and 5 differences between them will be explained and each difference separately.

* **Introduction**

An arrow function expression is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

Regular functions created using function declarations or expressions are constructible and callable. Since regular functions are constructible, they can be called using the new keyword.

However, the arrow functions are only callable and not constructible, i.e., arrow functions can never be used as constructor functions.

* **Subject** (Content)

**1. This value**

**1.1 Regular function**

Inside of a regular JavaScript function, this value (aka the execution context) is dynamic, the dynamic context means that the value of this depends on how the function is invoked.

**1.2 Arrow function**

The behavior of this inside of an arrow function differs considerably from the regular function's this behavior. The arrow function doesn't define its own execution context, no matter how or where being executed, this value inside of an arrow function always equals this value from the outer function. In other words, the arrow function resolves this lexically.

**2. Constructors**

**2.1 Regular function**

As seen in the previous section, the regular function can easily construct objects.

For example, the new Car() function creates instances of a car:

function Car(color) {

this.color = color;

}

const redCar = new Car('red');

redCar instanceof Car; // => true

Car is a regular function. When invoked with new keyword new Car('red') — new instances of Car type are created.

**2.2 Arrow function**

A consequence of this resolved lexically is that an arrow function cannot be used as a constructor.

If you try to invoke an arrow function prefixed with new keyword, JavaScript throws an error:

const Car = (color) => {

this.color = color;

};

const redCar = new Car('red');

Invoking new Car('red'), where Car is an arrow function, throws TypeError: Car is not a constructor.

**3. arguments object**

**3.1 Regular function**

Inside the body of a regular function, an argument is a special array-like object containing the list of arguments with which the function has been invoked.

Let's invoke myFunction() function with 2 arguments:

function myFunction() {

console.log(arguments);

}

myFunction('a', 'b'); // logs { 0: 'a', 1: 'b', length: 2 }

Inside of myFunction() body the arguments is an array-like object containing the invocation arguments: 'a' and 'b'.

**3.2 Arrow function**

On the other side, no arguments special keyword is defined inside an arrow function.

Again (same as with this value), the arguments object is resolved lexically: the arrow function accesses arguments from the outer function.

Let's try to access arguments inside of an arrow function:

function myRegularFunction() {

const myArrowFunction = () => {

console.log(arguments);

}

myArrowFunction('c', 'd');

}

myRegularFunction('a', 'b'); // logs { 0: 'a', 1: 'b', length: 2 }

The arrow function myArrowFunction() is invoked with the arguments 'c', 'd'. Still, inside of its body, arguments object equals to the arguments of myRegularFunction() invocation: 'a', 'b'.

**4. Implicit return**

**4.1 Regular function**

return expression statement returns the result from a function:

function myFunction() {

return 42;

}

myFunction(); // => 42

If the return statement is missing, or there's no expression after return statement, the regular function implicitely returns undefined:

function myEmptyFunction() {

42;

}

function myEmptyFunction2() {

42;

return;

}

myEmptyFunction(); // => undefined

myEmptyFunction2(); // => undefined

**4.2 Arrow function**

You can return values from the arrow function the same way as from a regular function, but with one useful exception.

If the arrow function contains one expression, and you omit the function's curly braces, then the expression is implicitly returned. These are the inline arrows function.

const increment = (num) => num + 1;

increment(41); // => 42

The increment() arrow consists of only one expression: num + 1. This expression is implicitly returned by the arrow function without the use of return keyword.

**5. Methods**

**5.1 Regular function**

The regular functions are the usual way to define methods on classes.

In the following class Hero, the method logName() is defined using a regular function:

class Hero {

constructor(heroName) {

this.heroName = heroName;

}

logName() {

console.log(this.heroName);

}

}

const batman = new Hero('Batman');

Usually, the regular functions as methods are the way to go.

Sometimes you'd need to supply the method as a callback, for example to setTimeout() or an event listener. In such cases, you might encounter difficulties accessing this value.

For example, let's use use logName() method as a callback to setTimeout():

setTimeout(batman.logName, 1000);

// after 1 second logs "undefined"

After 1 second, undefined is logged to console. setTimeout() performs a simple invocation of logName (where this is the global object). That's when the method is separated from the object.

Let's bind this value manually to the right context:

setTimeout(batman.logName.bind(batman), 1000);

// after 1 second logs "Batman"

batman.logName.bind(batman) binds this value to batman instance. Now you're sure that the method doesn't lose the context.

Binding this manually requires boilerplate code, especially if you have lots of methods. There's a better way: the arrow functions as a class field.

**5.2 Arrow function**

Thanks to Class fields proposal (at this moment at stage 3) you can use the arrow function as methods inside classes.

Now, in contrast with regular functions, the method defined using an arrow bind this lexically to the class instance.

Let's use the arrow function as a field:

class Hero {

constructor(heroName) {

this.heroName = heroName;

}

logName = () => {

console.log(this.heroName);

}

}

const batman = new Hero('Batman');

* **Conclusions**

Understanding the differences between regular and arrow functions helps choose the right syntax for specific needs, this value inside a regular function is dynamic and depends on the invocation. But this inside the arrow function is bound lexically and equals to this of the outer function.

arguments object inside the regular functions contains the list of arguments. The arrow function, on the opposite, doesn't define arguments (but you can easily access the arrow function arguments using a rest parameter ...rags).

If the arrow function has one expression, then the expression is returned implicitly, even without using the return keyword, Last but not least, you can define methods using the arrow function syntax inside classes. Fat arrow methods bind this value to the class instance, Anyhow the fat arrow method is invoked, this always equals the class instance, which is useful when the methods are used as callbacks.