Arrow Functions

This lesson covers the new way of declaring functions introduced in ES6.

WE'LL COVER THE FOLLOWING

What is an arrow function?

ES6 introduced fat arrows (`=>`) as a way to declare functions. This is how we would normally declare a function in ES5:

```
const greeting = function(name) {
  console.log("hello " + name);
}
```

The new syntax with a fat arrow looks like this:

```
var greeting = (name) => {
  console.log(`hello ${name}`);
}
```

We can go further if we only have one parameter. We can drop the parentheses and write:

```
var greeting = name => {
  console.log(`hello ${name}`);
}
```

If we have no parameters at all, we need to write empty parenthesis like this:

```
var greeting = () => {
  console.log(`hello ${name}`);
}
```

Implicitly return

With arrow functions, we can skip the explicit `return` and return like this:

```
const greeting = name => `hello ${name}`;
```

Look at a side by side comparison with an old ES5 Function:

```
const oldFunction = function(name){
  return `hello ${name}`
}

const arrowFunction = name => `hello ${name}`;
```

Both functions achieve the same result, but the new syntax allows you to be

Both functions achieve the same result, but the new syntax allows you to be more concise. Beware! Readability is more important than conciseness so you

might want to write your function like this if you are working in a team and not everybody is totally up-to-date with ES6.

```
const arrowFunction = (name) => {
  return `hello ${name}`;
}
```

Let's say we want to implicitly return an object literal. We'd do it like this:

```
const race = "100m dash";
const runners = [ "Usain Bolt", "Justin Gatlin", "Asafa Powell" ];

const results = runners.map((runner, i) =>  ({ name: runner, race, place: i + 1}));

console.log(results);
```

In this example, we're using the `map` function to iterate over the array `runners`. The first argument is the current item in the array, and the `i` is the index of it. For each item in the array we are then adding into `results` an Object containing the properties `name`, `race`, and `place`.

To tell `JavaScript` what's inside the curly braces is an **object literal** we want to implicitly return, so we need to wrap everything inside parentheses. Writing `race` or `race: race` is the same.

Arrow functions are anonymous

As you can see from the previous examples, arrow functions are **anonymous**.

If we want to have a name to reference them we can bind them to a variable:

```
const greeting = name => console.log(`hello ${name}`);

greeting("Tom");
```

Arrow function and the `this` keyword

You need to be careful when using arrow functions in conjunction with the `this` keyword, as they behave differently from normal functions.

When you use an arrow function, the `this` keyword is inherited from the parent scope.

This can be useful in cases like this one:

Output

JavaScript

HTML

CSS (SCSS)

This is a box

Console

Clear

The problem in this case is that the first `this` is bound to the `const` box but the second one, inside the `setTimeout`, will be set to the `Window` object, throwing this error:

```
Uncaught TypeError: cannot read property "toggle" of undefined
```

You won't be able to see this error in the Educative environment

You won't be able to see this error in the Educative environment.

Since we know that **arrow functions** inherit the value of `this` from the parent scope, we can re-write our function like this:

Output

JavaScript

HTML

CSS (SCSS)

This is a box

Here, the second `this` will inherit from its parent, and will be set to the `const` box.

Running the example code, you should see our `div` turning red for just half a second.

When you should avoid arrow functions

Using what we know about the inheritance of the `this` keyword we can define some instances where you should **not** use arrow functions.

The next two examples show when to be careful using `this` inside of arrows.

Example 1

```
const button = document.querySelector("btn");
button.addEventListener("click", () => {
  // error: *this* refers to the `Window` Object
```

```
  this.classList.toggle('on');
})
```

Example 2

```
const person1 = {
  age: 10,
  grow: function() {
    this.age++;
    console.log(this.age);
  }
}

person1.grow();
// 11

const person2 = {
  age: 10,
  grow: () => {
    // error: *this* refers to the `Window` Object
    this.age++;
    console.log(this.age);
  }
}

person2.grow();
```

Another difference between arrow functions and normal functions is the access to the `arguments object`. The `arguments object` is an array-like object that we can access from inside functions and contains the values of the arguments passed to that function.

A quick example:

```
function example(){
  console.log(arguments[0])
}

example(1,2,3);
// 1
```

As you can see we accessed the first argument using an array notation

`arguments[0]`.

Similarly to what we saw with the `this` keyword, arrow functions inherit the value of the `arguments object` from their parent scope.

Let's have a look at this example with our previous list of runners:

Output

JavaScript

HTML

CSS (SCSS)

```
const showWinner = () => {
  const winner = arguments[0];
  console.log(`${winner} was the winner`)
}

showWinner( "Usain Bolt", "Justin Gatlin", "Asafa Powell" )
```

Console

Clear

This code will return:

```
ReferenceError: arguments is not defined
```

To access all the arguments passed to the function we can either use the old function notation or the spread syntax (which we will discuss more in the lesson: Spread Operator and Rest Parameters)

Remember that `arguments` is just a keyword, it's not a variable name.

Example with **arrow function**:

```
const showWinner = (...args) => {
  const winner = args[0];
  console.log(`${winner} was the winner`)
}
```

```
showWinner("Usain Bolt", "Justin Gatlin", "Asafa Powell" )
// "Usain Bolt was the winner"
```

Example with **function**:

```
const showWinner = function() {
  const winner = arguments[0];
  console.log(`${winner} was the winner`)
}
showWinner("Usain Bolt", "Justin Gatlin", "Asafa Powell")
// "Usain Bolt was the winner"
```

In the next lesson, we 'll take a quiz and a coding challenge to test the concepts covered in this lesson.