Arrow Functions =>

An **arrow function** is a function that uses the **=>** symbol instead of the **function** keyword. Other things to know about arrow functions:

- **arrow functions return *implicitly***, which means that they always return a value--even when the **return** keyword is absent.
- **arrow functions** have concise sytnax:
  - **=>** is 2 characters, **function** is 8 characters
  - if there is only one line of logic, the curly braces **{ }** and **return** keyword may be omitted
  - if there is only one argument, the argument **( )** may be omitted.

Let's start by refactoring a "regular" function definition into an **arrow function**:

1. Define a "regular" function, which, though simple, has a paramter, so expects an input (argument) and also returns a value:

```
function greetUser(user) {
  return `Welcome back, ${user}!`;
}
```

hoising Function definitions are **hoisted**, meaning that they are lifted to the top of their **scope**, regardless of where they occur in the lines of code.

2. Call the function, supplying the expected argument. Since the function returns a value, set the call equal to a variable. This captures the return value; log the variable to see the greeting:

```
function greetUser(user) {
  return `Welcome back, ${user}!`;
}

let greeting = greetUser("Zebra123");
console.log(greeting); // Welcome back, Zebra123!
```

3. To verify that the function definition has been hoisted, call the function above the function definition; it still works:

```
  let greet = greetUser("Hoisted1");
  console.log(greet); // Welcome back, Hoisted1!

  function greetUser(user) {
    return `Welcome back, ${user}!`;
  }

  let greeting = greetUser("Zed1");
  console.log(greeting); // Welcome back, Zed1!
```

converting the function definition into an arrow function

4. Replace the word **function** with **const** or **let**. This turns **greetUser** into a variable declaration, so put an equal sign right after the name of the variable,:

```
const greetUser = (user) {
  return `Welcome back, ${user}!`;
}
```

5. Put the "fat arrow" **=>** to the right of the parentheses:

```
const greetUser = (user) => {
  return `Welcome back, ${user}!`;
}
```

6. Run the page. Now, the first function call--the one above the function--fails. This is because a fat-arrow function, set equal to a variable, is not hoisted. Comment out the first function call, so that it stops throwing the error.

making the fat-arrow function even simpler

7. Since there's only the one line of logic, we can make the fat-arrow function syntax even more concise. Get rid of the curly braces, the *return* keyword and the parentheses around the argument:

```
const greetUser = user => `Welcome back, ${user}!`;

console.log(greetUser('Zed1')); // Welcome back, Zed1!
```

It is even easier to convert a function expression into a fat-arrow function, since it a function expression is already set equal to a variable:

5. Start with a function expression:

```

```

```
const addNums = function(n1, n2) {
  return n1 + n2;
}

console.log(addNums(135, 246)); // 381
```

Convert the function expression to an arrow function:

6. Delete the word **function**:

```
const addNums = (a, b) {
    return a + b;
}
```

7. Add **=>** after the parentheses:

```
const addNums = (a, b) => {
    return a + b;
}
```

8. Since there is only one line of logic, delete **{}** and **return**. We have to leave the parentheses, because there is more than one argument

```
const addNums = (a, b) => a + b;

console.log(addNums(237, 365)); // 602
```

when you cannot simplify the arrow syntax too much
If there is more than one line of logic, as in this next fat-arrow function, you have to keep the **{}** and **return** keyword:

9. Declare a variable, set equal to a fat-arrow function. Have it peform a few lines of code, so that we must keep the **{}** and **return** keyword. However, since there is only one argument, omit the **()** around the argument:

```
const capitalizeWord = word => {
  let firstChar = word[0].toUpperCase()
  let restOfChars = word.slice(1); // isolate the rest of the word
  return firstChar + restOfChars; // concat and return the 2 parts
}

capit = capitalizeWord('elephant')
console.log(capit) // Elephant
```

arrow functions with map() and filter()

The concise syntax of the arrow function makes it well suited for deployment as the callback function of array methods, **map()** and **filter()** We will now do a map-filter chaining example, first as a "regular" function and then with arrow functons. The function specs are as follows:

- the function will take an array of strings and numbers as its argument
- the function will a new array, **nums**, containing just the numbers
- "number-like strings" are to be converted to real numbers and included in **nums**
- if the callback has only one callback argument, its parentheses can be omitted

First, we'll do the map-filter chaining version with the **function** keyword.

9. Declare an array,**mix**, of strings and numbers, including a few "number-like strings":

```
  const mix = ['appple', 3, 21, 'banana', 33, '55', 89, 'cherry', '63',
"77"];
```

10. Declare **nums**, set equal to **mix** with the **map()** method called on it and with its callback function with argument, **e**:

```
  const nums = mix.map(function(e) {
  });
```

11. Pass each item to the **Number()** method and return it:

```
  const nums = mix.map(function(e) {
    return Number(e);
  })
```

- if the item is already a number, passing it to the **Number()** method will have no effect.
- if the item is already a "number-like string", passing it to the **Number()** method return an actual number
- if the item is a string, such as "banana", passing it to the **Number()** method will return **NaN**, which is falsey.
- as we recall, falsey values return false in a boolean context, which means **NaN** will be filtered out by the **filter()** method

12. Chain **filter()** onto the end of **map()** to weed out the **NaN** values produced by the **Number()** method. Just return is **e**, the current item. The **NaN** items will return false and therefore be filtered out:

```
  const nums = mix.map(function(e) {
    return Number(e);
  }).filter(function(e) {
```

```
  return e;
});
```

13. Log **nums** to make sure it worked:

```
console.log(nums); // [3, 21, 33, 55, 89, 63, 77];
```

14. Now, try that again, but with arrow functions for callbacks. Just remove the word **function** and replace it with **=>** on the other side of the parentheses:

```
const numz = mix.map((e) => {
  return Number(e);
}).filter((e) => {
    return e;
});

console.log(numz); // [3, 21, 33, 55, 89, 63, 77];
```

15. Make the code more concise, as this is a hallmark of arrow functions:

- get rid of the curly braces and return keyword
- omit the callback argument parentheses, since there's just the one argument
- it is customary when writing such lean code to also lose the semi-colons
- as a final nod to minimalism, you can even make the array just one character:

```
const n = mix.map(e => Number(e)).filter(e => e);

console.log(n); // [3, 21, 33, 55, 89, 63, 77];
```

END Lesson 07.02
NEXT Lesson 07.03