Arrow Functions

Arrow functions provide two main benefits over regular functions. First, they're more terse. Second, they make managing the this keyword a little easier.

What I've seen with new developers learning about Arrow Functions is that it's not really the concept itself that's difficult to grasp. Odds are you're already familiar with functions, their benefits, use cases, etc. However, for some reason, it's the actual syntax that throws your brain for a loop when you're first exposed to them. Because of that, we're going to take things slow and first just introduce how the syntax compares with typical functions you're used to.

Here we have a very basic function declaration and a function expression.

```
// fn declaration
function add(x, y) {
  return x + y;
}

// fn expression
const add = function (x, y) {
  return x + y;
};
```

Now, if we wanted to change that function expression to an arrow function, we'd do it like this.

```
const add = function (x, y) {
  return x + y;
};

var add = (x, y) => {
  return x + y;
};
```

Again, the most difficult part about getting started with arrow functions is just getting used to the syntax. Once you're cool with it, move on and we'll dive deeper. At this point you may be wondering what all the hype is about with arrow functions. Truthfully, the example above doesn't really lend well to their strengths. What I've found is that arrow functions really thrive when you're using anonymous functions. We can warm our brain up a little more to the syntax by looking at another basic example of this is using .map.

```
users.map(function () {});

users.map(() => {});
```

Alright enough with the warm up. Let's dive into it.

Let's say we had a getTweets function that took in a user id and, after hitting a poorly designed API, returned us all of the user's Tweets with over 50 stars and retweets. Using promise chaining, that function may look something like this,

```
function getTweets(uid) {
  return fetch("//api.users.com/" + uid)
    .then(function (response) {
      return response.json();
    })
    .then(function (response) {
      return response.data;
    })
    .then(function (tweets) {
      return tweets.filter(function (tweet) {
        return tweet.stars > 50;
      });
    })
    .then(function (tweets) {
      return tweets.filter(function (tweet) {
        return tweet.rts > 50;
      });
    });
}
```

Well, it works. But it's not the prettiest function in the world 🤷. Even though this specific implementation is kind of dense, the idea is all too common. Let's take a look at how what we know about arrow functions thus far, can improve our getTweets function.

```
function getTweets(uid) {
  return fetch("//api.users.com/" + uid)
    .then((response) => {
      return response.json();
    })
    .then((response) => {
      return response.data;
    })
    .then((tweets) => {
      return tweets.filter((tweet) => {
        return tweet.stars > 50;
      });
    })
    .then((tweets) => {
      return tweets.filter((tweet) => {
        return tweet.rts > 50;
      });
    });
}
```

OK, cool. It looks basically the same we just didn't have to type function. Beneficial, but nothing worth Tweeting about. Let's look at the next benefit of arrow functions, "implicit returns". With arrow functions, if your function has a "concise body" (a fancy way for saying one line function), then you can omit the "return" keyword and the value will be returned automatically (or implicitly).

So the add example from earlier can be updated to look like this,

```
var add = function (x, y) {
  return x + y;
};

var add = (x, y) => x + y;
```

and more importantly, the getTweets example can be update to look like this,

```
function getTweets(uid) {
  return fetch("//api.users.com/" + uid)
    .then((response) => response.json())
    .then((response) => response.data)
    .then((tweets) => tweets.filter((tweet) => tweet.stars > 50))
    .then((tweets) => tweets.filter((tweet) => tweet.rts > 50));
}
```

Now we're talking 📈. That code is not only much easier to write, but more importantly, it's much easier to read.

Now, one further change we can make is that if the arrow function only has one parameter, you can omit the () around it. With that in mind, getTweets now looks like this,

```
function getTweets(uid) {
  return fetch("//api.users.com/" + uid)
    .then((response) => response.json())
    .then((response) => response.data)
    .then((tweets) => tweets.filter((tweet) => tweet.stars > 50))
    .then((tweets) => tweets.filter((tweet) => tweet.rts > 50));
}
```

Overall, I'd say that's a huge improvement in just about every category.

The next benefit of arrow functions is how they manage the this keyword. Let's take a look at some typical React code.

```
class Popular extends React.Component {
  constructor(props) {
    super();
    this.state = {
      repos: null,
    };

    this.updateLanguage = this.updateLanguage.bind(this);
  }
  componentDidMount() {
    this.updateLanguage("javascript");
  }
  updateLanguage(lang) {
    api.fetchPopularRepos(lang).then(function (repos) {
      this.setState(function () {
        return {
          repos: repos,
        };
      });
    });
  }
  render() {
    // Stuff
  }
}
```

When the component mounts, it's making an API request (to the Github API) to fetch JavaScript's most popular repositories. When it gets the repositories, it takes them and updates the component's local state, or at least that's what we want it to do. Unfortunately, it doesn't do that. Instead, we get an error. Can you spot the bug? The error the code above is going to throw is "cannot read setState of undefined".

```
class Popular extends React.Component {
  constructor(props) {
    super();
    this.state = {
      repos: null,
    };

    this.updateLanguage = this.updateLanguage.bind(this);
  }
  componentDidMount() {
    this.updateLanguage("javascript");
  }
  updateLanguage(lang) {
    api.fetchPopularRepos(lang).then(
      function (repos) {
        this.setState(function () {
          return {
            repos: repos,
          };
        });
      }.bind(this),
    );
  }
  render() {
    // Stuff
  }
}
```

This is the second major benefit as to why arrow functions are great, they don't create their own context. What that means is that typically the this keyword Just Works without you having to worry about what context a specific function is going to be invoked in. So by using arrow functions in the updateLanguage method, we don't have to worry about this which means we don't have to call .bind anymore.

```
updateLanguage(lang) {
  api.fetchPopularRepos(lang)
    .then((repos) => {
      this.setState(() => {
        return {
          repos: repos
        }
      });
    });
}
```

Nice to know-s

At this point, we've covered all of the "need to know-s" about arrow functions. There are, however, two different "nice to knows" that I think are worth mentioning.

Looking at the updateLanguage method again, if we wanted to implicitly return the object inside of the setState callback, how would we do that? Your first intuition would be to remove the return statement and just return an object.

```
api.fetchPopularRepos(lang).then((repos) => {
  this.setState(() => {
    repos: repos;
  });
});
```

The problem with this, as you probably guessed, is that that syntax is the exact same as creating a function body. JavaScript can't magically tell the difference between when you want to create a function body and when you want to return an object so it'll throw an error. To fix this, we can wrap the object inside of ().

```
api.fetchPopularRepos(lang).then((repos) => {
  this.setState(() => ({
    repos: repos,
  }));
});
```

Now, with that syntax, we can use an arrow function to implicitly return an object.

Now I know if I don't put this, someone will mention it. As a bonus since we're using ES6, we can go ahead use ES6's shorthand property and method names feature to get rid of the repos:repos and use Arrow Function's implicit return to shorten it up a bit.

```
api
  .fetchPopularRepos(lang)
  .then((repos) => this.setState(() => repos));
```

Next tip. Say we wanted to examine the previous state of the component inside of setState by logging it. If this was your setState function, how would you approach logging nextState?

```
this.setState((nextState) => ({
  repos: repos,
}));
```

The obvious move would be to change your implicit return to an explicit return, create a function body, then log inside of that body.

```
this.setState((nextState) => {
  console.log(nextState);
  return {
    repos: repos,
  };
});
```

Well, that's pretty annoying. There is a better way though and it's done using the || operator. Instead of messing with all of your code, you can do something like this

```
this.setState(
  (nextState) =>
    console.log(nextState) || {
      repos: repos,
    },
);
```

So clever.