COMP6080 Front-End Programming

Tutorial 10 — The End

Sidney Pham

sidney.pham@unsw.edu.au

UNSW

Week 10, Term 3 2020

The End

We've made it to the  of content!

The End

We've made it to the [illegible] of content! Personally, it's the last week of tutoring for me, which feels like it deserves reflection. Thanks for collectively being a really cool class!

The End

We've made it to the [illegible] of content! Personally, it's the last week of tutoring for me, which feels like it deserves reflection. Thanks for collectively being a really cool class!

Speaking of reflection, [illegible] is up — help us make this course better next time!

The End

We've made it to the [illegible] of content! Personally, it's the last week of tutoring for me, which feels like it deserves reflection. Thanks for collectively being a really cool class!

Speaking of reflection, [illegible] is up—help us make this course better next time!

Oh, also, I have no idea what happens to my slides on Gitlab once I graduate, maybe download them if you want to keep them.

The Tutorial

Tutorial 10 is an opportunity for the tutor and their students to highlight particular t students feel might need a bit more exploration, particula the exam that is coming up.

A number of topics h up that you may want your tutor to go through:

1. HTML
2. CSS — Selectors, Formatting, Layouts, Dev Tools
3. NodeJS — Basics
4. Javascript in browser — Importing, DOM manipulation, forms, events, `localStorage`
5. ReactJS — Components, hooks, routing
6. UI/UX principles
7. Accessibility principles
8. Testing principles

Giving You Some Thinking Time...

Please think of things you're a little confused about in this course!

Giving You Some Thinking Time...

Please think of things you're a little confused about in this course! But in the meantime, I just wanted to cover one fun thing today — closures!

TL;DR

You can think of a closure as a tuple: `(function, environment)`.

TL;DR

You can think of a closure as a tuple: `(function, environment)`.

An environment is a mappping from variables to values. That is, it's basically a function that gives you values for variables.

TL;DR

You can think of a closure as a tuple: `(function, environment)`.

An environment is a mappping from variables to values. That is, it's basically a function that gives you values for variables.

That's all the remembering you need to do, but it probably won't make much sense. Let's try and explain...

A Binary Adder

Let's create a functi[illegible] can use like `binaryAdd(2)(3) === 5`.

A Binary Adder

Let's create a function that we can use like `binaryAdd(2)(3) === 5`. An important thing to note is that we have first-class functions! (`binaryAdd` is a function that returns a function.)

A Binary Adder

Let's create a function that we can use like `binaryAdd(2)(3) === 5`. An important thing to note is that we have first-class functions! (`binaryAdd` is a function that returns a function.)


```
const binaryAdd = x => (y => x + y)
```


(The => operator is right-associative, so you don't actually need those parentheses; it's there to emphasise that we're returning a function.)

Using It

We can use it like:

```
const binaryAdd = x => y => x + y
const add2 = binaryAdd(2);
console.log(add2(5)) // 7
```

Using It

We can use it like:

```
const binaryAdd = x => y => x + y
const add2 = binaryAdd(2);
console.log(add2(5)) // 7
```


Calling `binaryAdd(2)` returns a function that takes a number `y` and returns `x + y`.

Using It

We can use it like:

```
const binaryAdd = x => y => x + y
const add2 = binaryAdd(2);
console.log(add2(5)) // 7
```


Calling `binaryAdd(2)` returns a function that takes a number `y` and returns `x + y`. But notice that `x` is only available within the scope of `binaryAdd`!

Using It

We can use it like:

```
const binaryAdd = x => y => x + y
const add2 = binaryAdd(2);
console.log(add2(5)) // 7
```


Calling `binaryAdd(2)` returns a function that takes a number `y` and returns `x + y`. But notice that `x` is only available within the scope of `binaryAdd`!

Calling `add2(5)` will bind `y` to 5, and then try to compute `x + 5`.

Using It

We can use it like:

```
const binaryAdd = x => y => x + y
const add2 = binaryAdd(2);
console.log(add2(5)) // 7
```


Calling `binaryAdd(2)` returns a function that takes a number `y` and returns `x + y`. But notice that `x` is only available within the scope of `binaryAdd`!

Calling `add2(5)` will bind `y` to 5, and then try to compute `x + 5`. This will try and find the value for `x` by performing scope resolution (checking the current environment, then all the outer environments).

Closures

What happens in languages with closures (like JavaScript) is that `binaryAdd(2)` returns a closure (i.e. `(function, environment)`), not just a function.

Closures

What happens in lan th closures (like JavaScript) is that `binaryAdd(2)` returns a closure (i.e. `(function, environment)`), not just a function. So, the lifetime of `x` is extended beyond the lifetime of `binaryAdd`.

Closures

What happens in lan th closures (like JavaScript) is that `binaryAdd(2)` returns a closure (i.e. `(function, environment)`), not just a function. So, the lifetime of `x` is extended beyond the lifetime of `binaryAdd`.

For example, we might do something like `window.onClick = () => alert(name);` Whenever `window.onClick` gets called, the value for `name` will be found in the closure!

That's Pretty Much It

It's honestly not the most important thing to know; it's hopefully already intuitively obvious.

That's Pretty Much It

It's honestly not the most important thing to know; it's hopefully already intuitively obvious. However, I think understanding *why* things work lets you talk about your code better (a very important thing).

There *is* More...

One thing to notice i sures kind of let us do encapsulation.

There *is* More...

One thing to notice is that closures kind of let us do encapsulation. Before, the `x` variable was available in the global scope, but we could run `add2(5)` successfully! We've hidden some state in our closure.

There *is* More...

One thing to notice is that closures kind of let us do encapsulation. Before, the `x` variable was available in the global scope, but we could run `add2(5)` successfully! We've hidden some state in our closure.

See the **Emulating private methods with closures** section of the MDN page for an example of how you can use functions as a simplified class!

There *is* More...

One thing to notice is that closures kind of let us do encapsulation. Before, the `x` variable was available in the global scope, but we could run `add2(5)` successfully! We've hidden some state in our closure.

See the **Emulating private methods with closures** section of the MDN page for an example of how you can use functions as a simplified class!

Essentially, our closure is *one* function with as much encapsulated state (the variables we close over) as we like.

Bonus: Currying

A common technique in functional programming (which you can totally do in JavaScript) is currying.

Bonus: Currying

A common technique in functional programming (which you can totally do in JavaScript) is currying.

Notice how we didn't do `const uncurriedAdd = (x, y) => x + y`, which would be called like `uncurriedAdd(2, 5)` instead of `binaryAdd(2)(5)`?

Bonus: Currying

A common technique in functional programming (which you can totally do in JavaScript) is currying.

Notice how we didn't do `const uncurriedAdd = (x, y) => x + y`, which would be called like `uncurriedAdd(2, 5)` instead of `binaryAdd(2)(5)`?

We've written a curried version, and now we can pass around partially applied functions. For example,

```
[1, 5, 2, 3].map(x => uncurriedAdd(2, x))
```

vs

```
[1, 5, 2, 3].map(binaryAdd(2))
```

Things to Cover

In the likely event th out of things to discuss, here are some ideas:

- Semantic HTML elements;
- Doing a random assignment 2 code review;

- Using a CSS framework like Bulma in React (rbx);

- Going through a lab question together;
- Trying to list UI/UX considerations on the fly;