COMP4021
Internet Computing

More on Promises

Gibson Lam

You have seen this before

Loading 5 Images

```
const myimage1 = new Image(),
      myimage2 = new Image(),
      myimage3 = new Image(),
      myimage4 = new Image(),
      myimage5 = new Image();
myimage1.onload = function() {
    myimage2.onload = function() {
        myimage3.onload = function() {
            myimage4.onload = function() {
                myimage5.onload = function() {

                    ... Now do something after all images are loaded...

                };
                myimage5.src = "myimage5.png";
            };
            myimage4.src = "myimage4.png";
        };
        myimage3.src = "myimage3.png";
    };
    myimage2.src = "myimage2.png";
};
myimage1.src = "myimage1.png";
```

- It has not included any error handling!

You have seen this before

How About Five Images?

```
const myimage1 = new Image(),
      myimage2 = new Image(),
      myimage3 = new Image(),
      myimage4 = new Image(),
      myimage5 = new Image();
myimage1.src = "myimage1.png";
myimage1.decode()
    .then(() => {
        myimage2.src = "myimage2.png";
        return myimage2.decode();
    })
    .then(() => {
        myimage3.src = "myimage3.png";
        return myimage3.decode();
    })
    .then(() => {
        myimage4.src = "myimage4.png";
        return myimage4.decode();
    })
    .then(() => {
        myimage5.src = "myimage5.png";
        return myimage5.decode();
    })
```

- This code loads five images one by one, and handles the error in one single `.catch()`

```
.then(() => {
    ...All images are loaded...
})
.catch((error) => {
    ...There is an error...
});
```

Simplifying the Code

You have seen this before

A function is created to load an image and return a promise

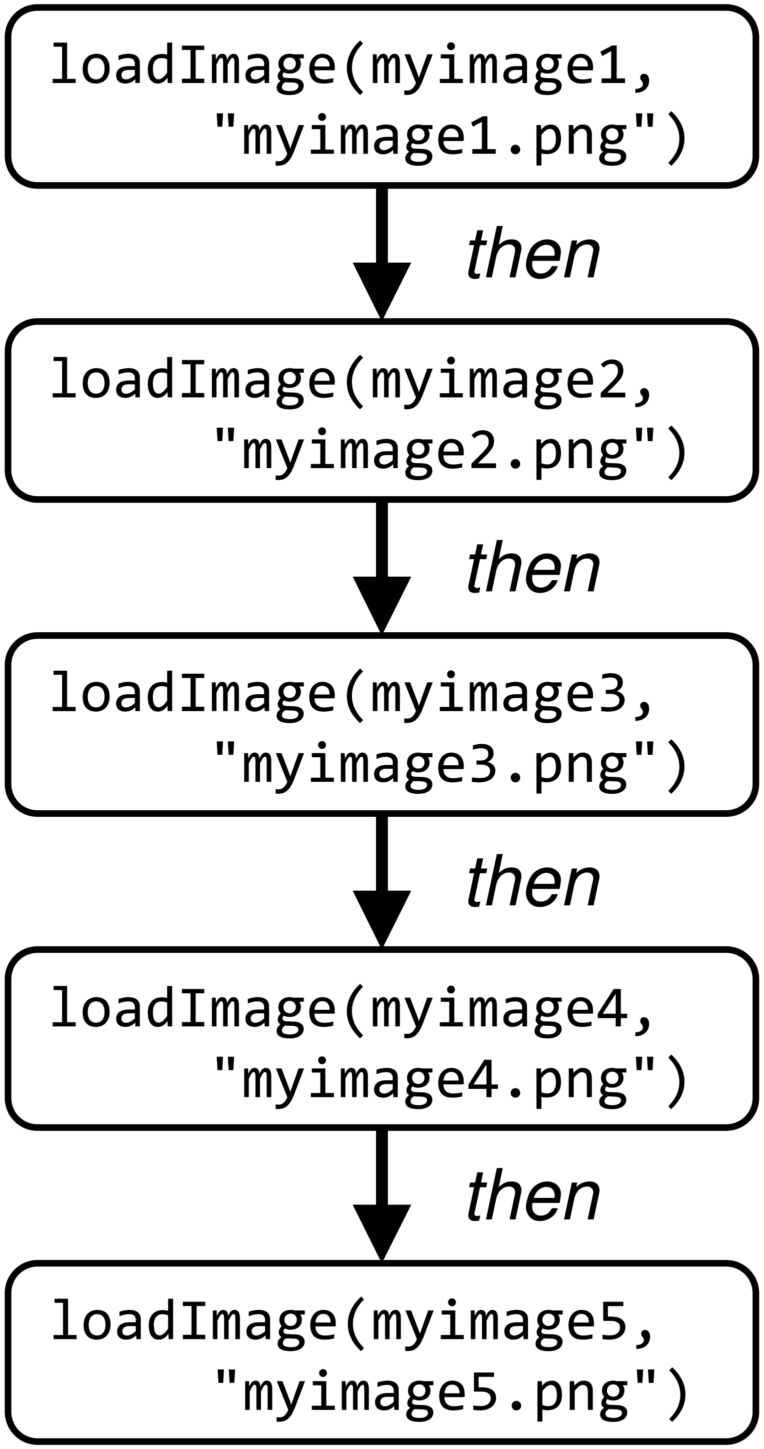
```
function loadImage(img, src) {
    img.src = src;
    return img.decode()
}
```

```
loadImage(myimage1, "myimage1.png")
    .then( () => loadImage(myimage2, "myimage2.png") )
    .then( () => loadImage(myimage3, "myimage3.png") )
    .then( () => loadImage(myimage4, "myimage4.png") )
    .then( () => loadImage(myimage5, "myimage5.png") )
    .then(() => {
        ...All images are loaded...
    })
    .catch((error) => {
        ...There is an error...
    });
```

Each promise from `loadImage()` *is returned by the arrow functions*

Running the Code

- Although the previous code runs asynchronously, it still does things one by one in the order shown on the right:
- This is done by using the promises appropriately



```
loadImage(myimage1,
     "myimage1.png")
```

```
loadImage(myimage2,
     "myimage2.png")
```

```
loadImage(myimage3,
     "myimage3.png")
```

```
loadImage(myimage4,
     "myimage4.png")
```

```
loadImage(myimage5,
     "myimage5.png")
```

Rewriting the Code

- You may be tempted to simplify the code to put it in a ‘synchronous’ way, like this:

```
loadImage(myimage1, "myimage1.png");
loadImage(myimage2, "myimage2.png");
loadImage(myimage3, "myimage3.png");
loadImage(myimage4, "myimage4.png");
loadImage(myimage5, "myimage5.png");
```

These run asynchronously

. . . *Now do something after all images are loaded* . . . ✗

- This code won’t work!

*This is wrong! This part likely runs **before** all images finished loading*

Using Async/Await

- If you want to simplify the code while maintaining the *finishing order*, you can make use of the `async` and `await` commands
- The `await` command forces you to wait for a promise to complete before continuing, i.e.:

```
await loadImage(myimage1,
                "myimage1.png");
```

...*Now do something* ***after image 1 is loaded***... ✓

Wait for the promise to finish before continuing

Running Promises in Order

- If you use `await`, this code will work in the order you want it to:

```
await loadImage(myimage1, "myimage1.png");
await loadImage(myimage2, "myimage2.png");
await loadImage(myimage3, "myimage3.png");
await loadImage(myimage4, "myimage4.png");
await loadImage(myimage5, "myimage5.png");
```

... *Now do something after all images are loaded* ... ✓

Code finishes one after another

- Then, does that mean promises are not asynchronous after using `await`?

Synchronous Or Asynchronous?

- Promises are still asynchronously run even if you use the `await` commands
- It simply makes this group of code to asynchronously run **together** in the given order

This code runs asynchronously

```
await loadImage(myimage1, "myimage1.png");
await loadImage(myimage2, "myimage2.png");
await loadImage(myimage3, "myimage3.png");
await loadImage(myimage4, "myimage4.png");
await loadImage(myimage5, "myimage5.png");
```

Async Functions

- JavaScript requires awaited promises to be put inside an ‘async’ function, i.e.:

```
async function loadAllImages() {
    await loadImage(myimage1, "myimage1.png");
    await loadImage(myimage2, "myimage2.png");
    await loadImage(myimage3, "myimage3.png");
    await loadImage(myimage4, "myimage4.png");
    await loadImage(myimage5, "myimage5.png");
}
```

- The above function runs its content async-hronously and implicitly returns a promise

Running an Async Function

- The function on the previous slide can only run asynchronously
- For example, if you run this code:

This is wrong again!

```
loadAllImages();
```

...*Now do something after all images are loaded*... ✗

- It won't work again because `loadAllImages()` run asynchronously!

The Proper Approach

- To wait for `loadAllImages()` to finish loading all images, you need to use promise again, as shown below:

This part now runs after all images are loaded

```
loadAllImages()
   .then(() => {
      ...Now do something
         after all images are loaded...
   });
```

✓

The Entire Code

```
function loadImage(img, src) {
    img.src = src;
    return img.decode()
}
async function loadAllImages() {
    await loadImage(myimage1, "myimage1.png");
    await loadImage(myimage2, "myimage2.png");
    await loadImage(myimage3, "myimage3.png");
    await loadImage(myimage4, "myimage4.png");
    await loadImage(myimage5, "myimage5.png");
}
loadAllImages()
    .then(() => {
        ...All images are loaded...
    });
    .catch((error) => {
        ...There is an error...
    });
```

Alternatively, this part can be put at the end of `loadAllImages()`

Example Use of Async/Await

- Async/await are commonly used, for example, in an Express server
- Remember in the lab, we have used the synchronous version of these functions:
  - `fs.readFileSync()`
  - `fs.writeFileSync()`
  - `bcrypt.hashSync()`
- These commands may affect the server performance as they synchronously block the server’s execution so it is not good

Using Asynchronous Code

- To improve the code, you can use their asynchronous version, i.e.:
  - `fs.readFile()`
  - `fs.writeFile()`
  - `bcrypt.hash()`

These two require the 'promise version' of `fs`

- An example server endpoint is shown on the next slide, which encodes the entire content of a file using hashing

Example Server Code

```
const fs =
  require("fs").promises;

app.get("/encode", (req, res) => {
  fs.readFile("message.txt")
    .then((content) => {
      return bcrypt.hash(content, 10);
    })
    .then((content) => {
      return fs.writeFile("secret.txt", content);
    })
    .then(() => {
      ...Job done!...
    });
});
```

A few `.then()` *have been used to run the code in an expected order before reaching this line of code*

Improved Server Code

- By using `async/await`, the code shown on the previous slide can become more concise

```
const fs = require("fs").promises;

app.get("/encode", async (req, res) => {
  let content = await
                fs.readFile("message.txt");
  content = await bcrypt.hash(content, 10);
  await fs.writeFile("secret.txt", content);

  ...Job done!...
});
```

- Much shorter code!