

COMP4021  
Internet Computing

# Using JavaScript Promises

Gibson Lam

# Using Event/Callback Functions

- JavaScript uses a lot of event/callback functions, e.g. when you load an image:

```
const myimage = new Image();
```

```
myimage.onload = function() {
```

```
    ...Do something when the image is loaded...
```

```
};
```

```
myimage.src = "myimage.png";
```

- You write the code like this because the load event works asynchronously

# The Running Order

- The actual order of running the code is:

- ① `const myimage = new Image();`
- ② `myimage.onload = function() {`
- ④ `...Do something when the image is loaded...`
- `};`
- ③ `myimage.src = "myimage.png";`

- Although the running order is a bit ambiguous, you probably think it is simple enough when only one image is loaded

# Loading Multiple Images

- You may use the same code twice to load two images on a web page:

*Loading image 1* { `const myimage1 = new Image();`  
`myimage1.onload = function() {`  
`...Do something when image 1 is loaded...`  
`};`  
`myimage1.src = "myimage1.png";`

*Loading image 2* { `const myimage2 = new Image();`  
`myimage2.onload = function() {`  
`...Do something when image 2 is loaded...`  
`};`  
`myimage2.src = "myimage2.png";`

# Issues With the Previous Code

- You probably notice that there are two issues with the previous code
  1. You cannot tell which image is loaded first, because the two images are loaded asynchronously
  2. The code is not suitable when you want to run some code after **BOTH** images are successfully loaded

# Cascading the Events

- If you want to load the images in the order you want, you will need to ‘cascade’ the events (i.e. put one inside another), like this:

```
const myimage1 = new Image(),
      myimage2 = new Image();
myimage1.onload = function() {
  myimage2.onload = function() {
    ... Now do something after both are loaded ...
  };
  myimage2.src = "myimage2.png";
};
myimage1.src = "myimage1.png";
```

For image 1

For image 2

# Things Can Get Messy

- Imagine that you have 5 images that you need to load and then want to run some code after **all of them** finished loading
- You would need to go four levels into the event functions, as shown in the example on the next slide
- In addition, if you want to handle the error event ( `.onerror()` ) for each image at the same time, it will quickly get out of control

# 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!



# Using Promises

- You can do some clever programming to simplify the code, such as keeping track of the number of images loaded
- Alternatively, you can use *promises*, which is a JavaScript object that makes asynchronous programming easier
- Although you can create your own promise object, in this presentation, we will mainly look at how to use the promise objects returned by some functions

# Promises From Functions

- Some JavaScript objects/modules contain functions for working with promises only
- For example, instead of using `.onload()` and `.onerror()` of an image object, you can run `.decode()` to get back a promise object:

```
const mypromise = myimage.decode();
```

 *This is a promise object*

# Using a Promise

- The promise object gives you two functions:
  - `mypromise.then(...)`
    - You use this function to run some code when the promise is **successful**
  - `mypromise.catch(...)`
    - You use this function to run some code when the promise has **failed**
- You need to keep in mind that the above functions run their code asynchronously

# The Promise From an Image

- For the promise object returned by `.decode()` of an image object:
  - You use the `.then()` function to do the work that was done by `.onload()`
  - Similarly, you use the `.catch()` function to handle the error that was originally handled by `.onerror()`
- An example is shown on the next slide

# Example of Using Promise

- Here is the code that uses a promise:

```
const myimage = new Image();
myimage.onload = function() {
    ...Do something when the image
        is loaded...
};
myimage.src = "myimage.png";
```

*Original code*

```
const myimage = new Image();
myimage.src = "myimage.png";
```

```
const mypromise = myimage.decode();
mypromise.then(() => {
    ...Do something when
        the image is loaded...
});
```

*Run this  
code after  
the image  
is loaded*

# Handling Error

- You can handle the error too by doing this:

...

```
const mypromise = myimage.decode();
```

...

*This is an Error object*



```
mypromise.catch((error) => {  
    ...Do something when there is an error...  
});
```

*This part handles any error  
when loading the image*

# Using Chaining

- You can make things more efficient by using promise *chaining*
- It allows you to do things in one JavaScript statements, instead of multiple statements

```
const myimage = new Image();  
myimage.src = "myimage.png";  
myimage.decode()  
    .then(() => {  
        ...Do something when image is loaded...  
    })  
    .catch((error) => {  
        ...Do something when there is an error...  
    });
```

*This is  
one single  
JavaScript  
sentence*

# How About Two Images?

- You can chain two promises together:

```
const myimage1 = new Image(),  
      myimage2 = new Image();  
myimage1.src = "myimage1.png";  
myimage1.decode()
```

*This works for the first image promise*

```
.then(() => {  
    myimage2.src = "myimage1.png";  
    return myimage2.decode();  
})
```

*You must return the promise!*

*This works for the second image promise*

```
.then(() => {  
    ...Both images are loaded...  
})
```

*This works for both promises*

```
.catch((error) => {  
    ...There is an error...  
});
```



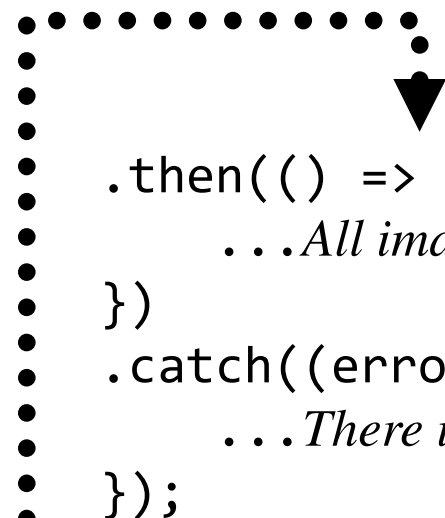
```

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();
  })

```

# How About Five Images?

- 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...
    });

```

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

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

## Simplifying the Code

```
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 any error...  
});
```

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