

COMP4021
Internet Computing

More on JavaScript

Gibson Lam

More JavaScript Features

- You have used some basic JavaScript
- In this presentation, we will look at some additional features, including:
 - More on declaring variables
 - Variable scopes
 - JavaScript arrays and objects
 - Spreading and destructuring
 - Arrow functions
 - Closure

More on Declaring Variables

- You have so far used the `let` keyword to declare variables, i.e.:

```
let luckyNumber = 7;
```

- Apart from `let`, you have two other ways to declare a variable in JavaScript:
 - Using `const`
 - Using `var`


Using const

- The `const` keyword, as its name suggests, gives you a constant, e.g.:

```
const luckyNumber = 7;
```

- That means the variable is read-only, you cannot reassign a value to the variable; doing so will give you an error, e.g.:

```
const luckyNumber = 7;  
luckyNumber = 8; // ERROR!
```



```
► Uncaught TypeError: Assignment to constant variable.
```

Using `let` or Using `const`

- Which one should you use?
 - Either one is fine!
- You can use `let`, instead of using `const`, in your entire program and nothing is going to break
- However, using `const` gives you early bug detection if your variable is indeed read only and should not be changed

Using var

- The var keyword is the ‘old-school’ approach, i.e.:

```
var luckyNumber = 7;
```

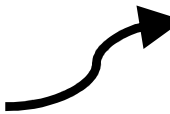
- Although it works fine, there are potential problems associated with it, such as variable redeclaration and confusing variable scopes
- You are recommended to use let and const in your work

Variable Scopes

- In most programming languages, you have different variable scopes when you create variables in different places
- In JavaScript, you have two basic variable scopes:
 - Global variable scope for those created outside of any function
 - Block variable scope for those created inside a block, i.e. within a pair of curly braces `{ ... }`, or in for loops

Variable Scope Examples

- Here is an example with a global variable, luckyNumber and a block variable, number

Both global and block variables can be used here 

```
<script>
let luckyNumber = 7;
for (let number = 0; number < 10; number++) {
  if (luckyNumber == number) { // Both okay
    alert("I have found your number!");
  }
}
alert("Your lucky number is " + number); // Wrong
</script>
```

Cannot access the block variable outside of the block

► Uncaught ReferenceError: number is not defined

Variables Without Declaration

- Although we have discussed the use of `let`, `const` and `var`, you can create variables without using any of them, e.g.:

<pre><script> username = "pikachu"; </script></pre>	<i>A global variable is created here if the variable has not been defined before</i>
---	--


- In this situation, the variable is created as a global variable (even when the variable is created inside functions!)

JavaScript Arrays

- JavaScript arrays, like other programming languages, are collections of things
- You can store different things inside an array, like this:

```
const prices = [5.5, 10, "FREE", 8];
```

- They have zero-based index and can be changed at any time, e.g.:



```
prices[1] = 8.5;
```

This changes the second item from 10 to 8.5

Some Array Functions

- You get an array length using `.length`, i.e.:
`console.log(prices.length); // 4`
- You can also change an array using `.splice()`
 - Adding a new item, e.g.:
`// Insert 6 at index 1`
`prices.splice(1, 0, 6);`
 - Removing an item, e.g.:
`// Remove item at index 3`
`prices.splice(3, 1);`

Going Through an Array

- You can use a typical for loop to go through an array, e.g.:

```
for (let i = 0; i < prices.length; i++) {  
    console.log(prices[i]);  
}
```



5.5
10
FREE
8

- Or, you can use for...of, e.g.:

```
for (const price of prices) {  
    console.log(price);  
}
```



You can use const in a for...of loop

JavaScript Objects

- You can think of JavaScript objects as associative arrays, i.e. collections of key/value pairs, also called properties
- You create a JavaScript object using curly braces { }, e.g.:

```
const myboyfriend = {  
    name: "Lok Man",  
    height: 175, weight: 65  
};
```

Getting Properties from Objects


- You can get properties from a JavaScript object in two ways
- First, you can use the dot (.):

```
console.log(myboyfriend.name);
```

- Or, you can use the key as an index:

```
console.log(myboyfriend["name"]);
```

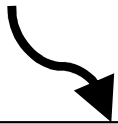
*Both of
them show
'Lok Man'*



Going Through an Object

- You can also loop through an object using `for...in`, e.g.:

```
for (const key in myboyfriend) {  
    console.log(key, ":",  
                myboyfriend[key]);  
}
```



```
name : Lok Man  
height : 175  
weight : 65
```

The Spread Operator

- The spread operator (...) is very useful for handling arrays and objects
- The operator 'lists' out items inside an array or an object, for example:

```
const prices = [5.5, 10, "FREE", 8];  
console.log(...prices);
```



5.5 10 'FREE' 8

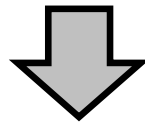
The diagram illustrates the spread operator's function. An arrow originates from the `...prices` portion of the `console.log(...prices);` line in the code block above and points to a rectangular box. Inside this box, the elements of the `prices` array are listed as arguments: `5.5`, `10`, `'FREE'`, and `8`.

- This operation is also called unpacking

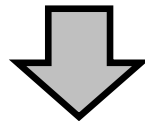
Unpacking Items

- You can think of unpacking as ‘removing the brackets’ and listing things separated by commas

```
prices = [5.5, 10, "FREE", 8]
```



```
...prices
```




```
5.5, 10, "FREE", 8
```

Using Spread for Arrays

- You can use the spread operator to copy the content of an array, e.g.:

```
const mycopy = [...prices];
```

 *This has the same content as prices after running this code*

- You can also easily extend an array:

```
const newprices = [...prices, 7, 9.5];
```

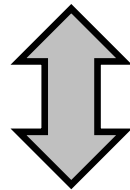
 *This becomes*
`[5.5, 10, "FREE", 8, 7, 9.5]`

Using Spread for Objects

- The spread operator is also handy for objects
- You can similarly extend an object like this:

```
const bf = { ...myboyfriend,  
             group: "Mirror" };
```

The object we have seen before



```
const bf = { name: "Lok Man",  
             height: 175, weight: 65,  
             group: "Mirror" };
```

Using Variables in Objects

- A more flexible way to manage objects is to use variables directly
- Here is an example to do the same thing as the code on the previous slide:

```
const group = "Mirror";  
const bf = {...myboyfriend, group};
```

This property uses the variable name as key and the variable content as the value



Getting Things Into Variables

- Let's consider this object:

```
const members = {  
  kids: 32,  
  teenagers: 20,  
  adults: 8 };
```

- You can use separate variables to store the properties from the object like this:

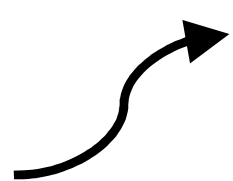
```
const kids = members.kids;  
const teenagers = members.teenagers;  
const adults = members.adults;
```

Using Destructuring

- The code near the bottom of the previous slide looks clumsy

- With destructuring, you can make it more concise, i.e.:

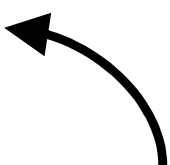
```
const { kids, teenagers, adults } = members;
```

 kids = 32
teenagers = 20
adults = 8

- Destructuring helps to unpack the object properties (or array values) into separate variables

Making New Objects

- You can use destructuring to rearrange your object content in a flexible way
- Here is an example:

```
const { kids, teenagers } = members;  
const youngMembers =  
    { kids, teenagers }; 
```

*This creates a new object
with two properties from
the variables*

More on Functions

- In JavaScript, functions are entities that can be stored in variables
- Apart from the traditional way of creating a function, you can also do this:

```
const myfunc = function(...) {  
    ...  
}
```


Arrow Functions

- You can also define a function in the minimal way, called *arrow functions*, e.g.:

```
const myfunc = (... ) => { ... }
```

- Arrow functions shorten the code especially when used as a callback, i.e.:

```
setTimeout(() => {alert("Wake up!");},  
           1000);
```

Arrow Functions with Return

- On the right is a function with a single return statement

```
function double(x) {  
    return 2 * x;  
}
```

- You can use an arrow function to remove the return statement, like this:

```
const double = (x) => 2 * x;
```

- And even the parentheses too:

```
const double = x => 2 * x;
```

Functions Inside Functions

- You can define functions (inner functions) inside another function (outer function), e.g.:

Outer function

```
function fullname(first, last) {
```

Inner function

```
    function formatName(sep) {  
        return last + sep + first;  
    }
```


```
    return formatName(", ");
```

```
}
```

Variable Access for Inner Function

Inner function

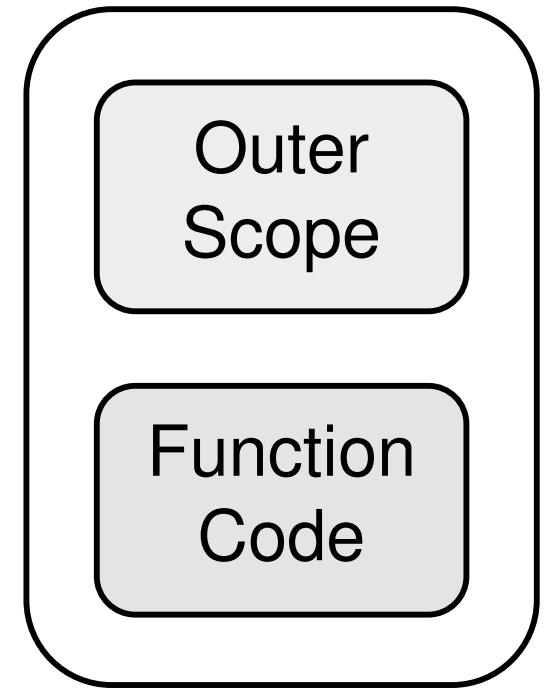
```
function formatName(sep) {  
    return last + sep + first;  
}
```



- In the previous example, you can see that the inner function can use the variables in the outer function
- Not only that, those variables are indeed ‘permanently bound’ together with the inner function

Closure

- After a function is created, it is stored as a JavaScript object
- As you would expect, this object should contain the code of the function
- In addition, the object also stores a reference to the outer scope, i.e. outer variables
- This is called a *closure*



The function object of an inner function

Variable Scope in a Closure

- In other programming languages, when a function finishes, its associated variable scopes are typically removed from memory
- However, JavaScript does not always work this way
- If a function has an inner function and the inner function still exists AFTER the outer function finishes, the variable scope of the outer function will not be removed

An Example Using Closure

- Here is another example of a closure with an inner function
- The inner function is an anonymous function that gets returned by the outer function:

```
function greeting(name) {  
    return function() {  
        console.log("Hi, " +  
                    name + " !");  
    };  
}
```

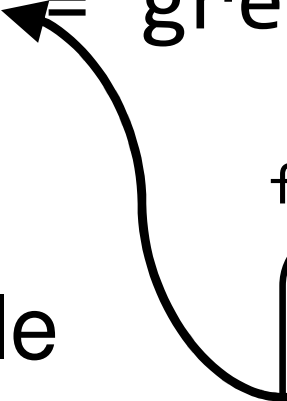
Running the Example Function

- You run the example function like this:

```
const sayHi ← greeting("Peter");
```

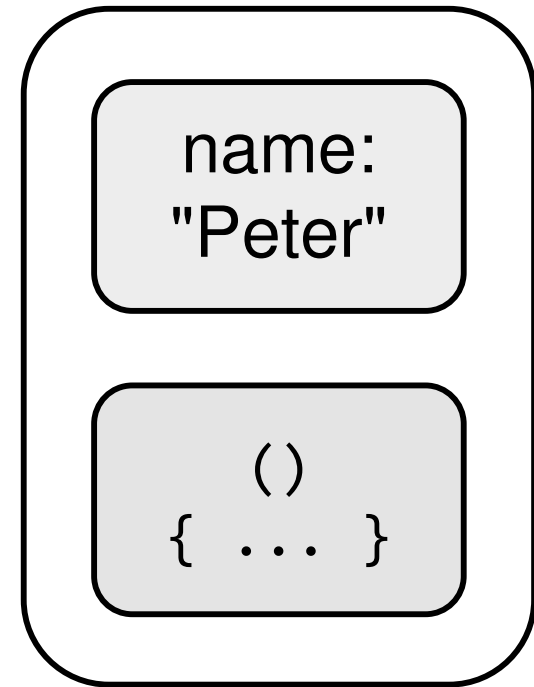
- You can see that the variable `sayHi` contains the inner function returned by `greeting()`

```
function greeting(name) {  
  return function() {  
    console.log("Hi, " +  
      name + "!");  
  };  
}
```



Content of the Inner Function

- Although `greeting()` has finished, the inner function, i.e. `sayHi`, still exists
- Therefore, the outer scope is not destroyed and is still referred to by the `sayHi` function



The sayHi function

Running the Inner Function

- If you run the inner function like this:

`sayHi();`

you will
see

`'Hi, Peter!'`
shown in the
console

```
function() {  
    console.log("Hi, " +  
                name + "!");  
}
```

*The content of
the variable is
'Peter'*

name:
"Peter"

The Use of Closure

- The outer scope effectively has become a private space for the inner function in a closure
- Then it can be used in various ways:
 - As a simple object with private data
 - As a function with states
 - As a function factory
 - As an extensible module
- We may look at some of these later