EAST WEST UNIVERSITY

# CSE479
## Web Programming

**Nishat Tasnim Niloy**

Lecturer

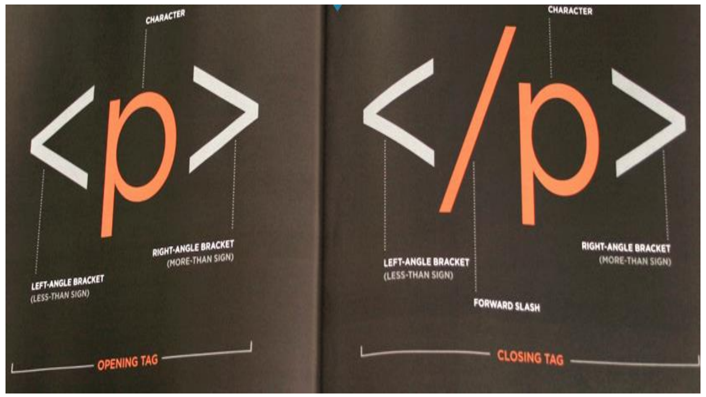
Department of Computer Science and Engineering

Faculty of Science and Engineering

## Topic 4

JavaScript Functions

## By the end of this unit you should be able to...


CHARACTER
<p>
LEFT-ANGLE BRACKET (LESS-THAN SIGN)
RIGHT-ANGLE BRACKET (MORE-THAN SIGN)
OPENING TAG
CHARACTER
</p>
LEFT-ANGLE BRACKET (LESS-THAN SIGN)
FORWARD SLASH
RIGHT-ANGLE BRACKET (MORE-THAN SIGN)
CLOSING TAG

- ❑ Write JavaScript functions that return one or more values
- ❑ Create anonymous functions in JavaScript using function expressions
- ❑ Immediately invoke function expressions
- ❑ Create and invoke arrow functions in JavaScript
- ❑ Describe how memory and variables work in JavaScript
- ❑ Explore JavaScript function calling patterns
- ❑ Explore JavaScript function calling patterns
- ❑ Create and use JavaScript closures
- ❑ Define and invoke functions that take optional parameters
- ❑ Implement and use classes in ES6
- ❑ Modify properties in JavaScript objects
- ❑ Demonstrate the use of class inheritance

## Getting a value from a function

```
function getRectangleArea(width, height) {
    const area = width * height;
    return area;
}

const firstRectangleArea = getRectangleArea(4, 5);
const secondRectangleArea = getRectangleArea(20, 6);
```

width, height parameters. Behave like local variables

arguments, values for the parameters

The ***getRectangleArea*** function is created using function declaration and returns a single value

Some functions return info to code that call them

## Getting multiple values from a function

```
function getCircleSizes(radius) {
    const area = Math.PI * Math.pow(radius, 2);
    const perimeter = 2 * Math.PI * radius;
    return [area, perimeter];
}
```

area is first
item in array

```
const circleArea = getCircleSizes(10)[0];
const circlePerimeter = getCircleSizes(10)[1];
```

The ***getCircleSizes*** function is created using function declaration and returns multiple values in an array

## Anonymous functions and function expressions

Expressions produce a value. They can be used where variables are expected.

If a function is placed where an expression is expected, it gets treated as an expression ⇒ **function expression**.

```
const areaFunction = function (width, height) {
    const area = width * height;
    return area;
}; // function is NOT processed until interpreter gets to this statement

const rectangleArea = areaFunction(7, 8);
```

## Anonymous functions and function expressions (2)

```
const area = function (width, height) {
   const area = width * height;
   return area;
};

const rectangleArea = area(7, 8);
```

Note that the function above has no name.

Functions with no names are called anonymous functions.

## Immediately invoked function expressions (IIFE) ‘iffy’

```
const recWidth = 99;
const recHeight = 4;

const area = (function(width, height) {
   return width * height;
}(recWidth, recHeight));
```

**Final parentheses** tell the interpreter to call the function immediately

**Grouping operator** is there to ensure that the interpreter treats this as an expression

## Arrow function expressions and lexical this

Arrow function expressions provide a shorthand notation for anonymous functions.

Referred to as ***lambda expressions*** in other languages (e.g., C#, Java8)

Arrow functions add **lexical scoping** for the `this` variable. **(A good thing!)**

**Lexical scoping**: a.k.a. **static scoping**, says that a variable can ONLY be referenced in the block of code in which it is defined.

JavaScript uses dynamic scoping by default.

**Dynamic scoping**: defines global variables that can be called or referenced from anywhere after being defined

## Arrow function expressions

General syntax, **parentheses required for multi-line arrow function**

```
(ar1, arg2) => { function body; return statement;}
```

No { } if **single-expression-function-body**
No return statement if **single-expression-function-body** returns a value --- return statement is implicit

```
const sum = (arg1, arg2) => arg1 + arg2;
```

Use empty parentheses if arrow function has no arguments

```
() => { function body; return statement;}
```

Parentheses optional if arrow function has only one argument

```
arg1 => { function body; return statement;}
```

## When to use anonymous functions, IIFEs, arrow functions

They are used for code that should be run once within a task.
**Note**: use arrow functions where possible.

Examples:

- ★ *As an argument when a function is called (to calculate a value for that function)*
- ★ *To assign the value of a property to an object*
- ★ *In event handlers and listeners (to perform a task when an event occurs)*
- ★ *To prevent name collision between two scripts using the same variable names*
- ★ *To wrap a set of code so that variable names in that code block have local scope (a.k.a function scope)*

## Variable scope - local, when using var => avoid var

Variable scope is the location in the code where the variable can be used.

Local variables with var are function-level variables:

- ★ **Must** be created ***inside*** a function, can use ***var*** keyword, but **avoid**
- ★ Can only be used inside the function in which they are created
- ★ Can be used from point of declaration to ***end of function***
- ★ Have local scope or ***function-level scope. Why?***
- ★ Hoisting: JavaScript moves all var declared variables to top of function

```
function a() {
   var hi = 'hi';
   console.log(hi); //hi
   console.log(bye); //undefined

   if (true) {
      var bye = 'bye';
      console.log(hi); //hi
      console.log(bye); //bye
   }
   console.log(hi); //hi
   console.log(bye); //bye
}
```

## Variable scope - global

Global variables:

- ★ Are script-level variables
- ★ Are typically declared outside a function
- ★ Are stored in browser memory for as long as the web page is loaded. **Important:** *local variable are destroyed when function returns*

- ★ If you declare a local variable anywhere **without the var, let, or const keyword**, it will be **treated as a global variable**. **Bad practice!**

- ★ **Can cause name collisions:**
  - ○ If a web page loads 2 scripts and they both have a global variable with the same name.
  - ○ Can result in errors

## For Personal Interest

- ★ Check out this fun demo of a front-end application https://www.strml.net/
- ★ Look at some cool “explorable explanations”
  - ○ https://explorabl.es/
  - ○ One of my favorites: https://ncase.me/trust/
- ★ If anything inspires you it might be a good idea for a project concept

## Function invocation patterns

The manner in which a function is called has a huge impact on how the code within it operates

★ Primarily on how the ***this*** parameter is established

4 ways to invoke a function

★ ***As a function***, invoked in a straightforward manner

★ ***As a method***, which ties the invocation to an object, enabling OOP

★ ***As a constructor***, in which a new object is created

★ ***Via its apply() or call() methods***, which can be complicated

## From arguments to parameters

If function is called with a different number of arguments than parameters, this is **not** an error.

If ***No. of arguments > than there are parameters***,

- ★ excess arguments are just **not** assigned to named parameters
- ★ How then do we access them?

If ***No. of arguments < than there are parameters***,

- ★ parameters with no corresponding arguments set to ***undefined***

Interestingly, all function invocations passed 2 implicit parameters

- ★ ***this***: function context → an object associated with invocation
- ★ ***arguments***: the collection of all the arguments passed to the function

## arguments implicit parameter

The collection has a ***length*** property

***arguments[1]*** accesses the second parameter, etc

It’s an **array-like** structure, but **not** an array

Array methods won’t work on it

It’s an implicit parameter because it is not explicitly listed

## “this” implicit parameter

An **object** that’s implicitly associated with the function invocation

It’s called the ***function context***.

What the ***this*** parameter points to is not necessarily the same as in Java

***this*** is **not** always defined by how a function is declared, but by how it is invoked

It is important to note how the value of ***this*** is determined for each type of invocation

## Invocation as a function

```
function myFunction(value) {
    console.log("Hello! " + value);
    console.log("How are you?");
    console.log(this);
}

myFunction(“teacher”);
```

- ★ ***this*** (the function context) refers to the global context - the ***window*** object
- ★ The ***window*** is implicitly the **owner** of the function
- ★ Note that this function is **not a method of an object**

## Example of function invocation

```
const person = {
    name: 'Calvin',
    age: 25,
    greet: function () {
        alert('My name is ' + this.name + '.');
    }
};
```

```
  // Add a new method to person
person.calculateAge = function (yearsFromNow) {

   // Why self?
   const self = this;

   function yearsOld() {
             // What if this.age instead of self.age?
      return self.age + yearsFromNow;
   }

   alert('I will be ' + yearsOld() + ' years old ' +
      yearsFromNow + ' years from now.');
};

person.calculateAge(10);
```

**this** is bound to or refers to the ***person*** object.

## Invocation as a method

```
const person = {
    name: 'Calvin',
    age: 25,
    greet: function () {
        alert('My name is ' + this.name + '.');
    }
};
```

***person.greet();***

- ★ A method is a special function that belongs to an object and is assigned to a property of that object
- ★ The object to which the method belongs is available within the body of the method as ***this***
- ★ ***this***, here refers to the person object

## Function invocation patterns

The manner in which a function is called has a huge impact on how the code within it operates

- ★ Primarily on how the ***this*** parameter is established

4 ways to invoke a function

- ★ ***As a function***, invoked in a straightforward manner
- ★ ***As a method***, which ties the invocation to an object, enabling object oriented programming
- ★ ***As a constructor***, in which a new object is brought into being
- ★ ***Via its apply() or call() methods***, which is somewhat complex (not today!)

## (Review) Invocation as a function

```
function myFunction(value) {
    console.log("Hello! " + value);
    console.log("How are you?");
}

myFunction("teacher");
```

- ★ ***this*** (the function context) refers to the global context - the ***window*** object
- ★ The ***window*** is implicitly the **owner** of the function
- ★ Note that this function is **not a method of an object**

## (Review) Invocation as a method

```
let person = {
    name: 'Calvin',
    age: 25,
    greet: function () {
        alert('My name is ' + this.name + '.');
    }
};
```

***person.greet();***

- ★ A method is a special function that belongs to an object and is assigned to a property of that object
- ★ The object to which the method belongs is available within the body of the method as ***this***
- ★ ***this***, here refers to the person object

## Invocation as a constructor (use classes instead)

In JavaScript, functions can be invoked with the ***new*** prefix similar to the way objects are constructed in other languages.

When this happens, ***this*** is bound to the new object.

Functions that are designed to be called by ***new*** are called ***constructor functions***.

Common practice is to **capitalize** these functions as a reminder to call them with ***new***.

## Constructor function invocation example (before classes)

```
function Person(first, last) {    // Person is a constructor function
    this.first = first;
    this.last = last;
    this.fullName = function () {
        return this.first + ' ' + this.last;
    };
    this.fullNameReversed = function () {
        return this.last + ', ' + this.first;
    };
}
const simon = new Person("Simon", "Willison");
```

NB: It is ***new*** that creates a new object, then calls ***Person()*** and sets ***this*** to refer to the newly created object.

## Prototype property

Every function or object in JavaScript has an implicit ***prototype*** property.

The ***prototype*** property is an object.

Each inherits methods and properties from its ***prototype***.

The prototype property allows you to add new properties to an existing prototype.

***Person.prototype.nationality = "English";***

## Adding methods to prototype property

```
Person.prototype.greet = function () {
   return this.first + ' says hi.';
};

console.log(new Person('Delvin', 'Thomas').greet()); //Delvin says hi.
```

Notice the ***greet*** function uses ***this*** to access the name property. ***this*** is bound to the ***Person*** prototype.

## Invocation using apply()

As a **functional object-oriented language**, JavaScript makes it possible for **functions to have methods** as well.

The ***apply*** function is a method on the ***Function.prototype*** – the prototype for all JavaScript functions.

***apply*** makes it possible to **use one object's method in the context of another**.

## Passing the correct arguments to apply()

Supply to the ***apply*** method an array with the correct number of arguments and the **object** to which ***this*** will be bound, also known as the ***function context***.

***apply*** can thus take two arguments:

1. a context for ***this*** and
2. **an array of arguments** that will be used for the method at hand

```
const calvin = new Person('Calvin', 'James');
const hobbes = {first: 'Hobbes'};
alert(calvin.greet.apply(hobbes)); //Hobbes says hi.
```

Even though ***hobbes*** does not have a ***greet*** method, we can still apply the ***greet*** method from ***calvin*** because ***hobbes*** has a ***first*** property.

## TODO

```
Person.prototype.greetFriends = function (friendA, friendB) {
   return this.first + ' says hi to ' + friendA.first + ' and ' + friendB.first + '.';
};

const bill = {first: 'Bill', last: 'Watterson'};
```

Use the apply invocation pattern to have calvin invoke the method **greetFriends()** so that the result is

**Bill says hi to Calvin and Hobbes.**

Work with your in-class partner on this.

## Invocation using call()

***call*** works like ***apply***, except that the arguments are passed in **directly** rather than as an array.

```
function avg() {
    let sum = 0;
    for (let i = 0, j = arguments.length; i < j; i++) {
        sum += arguments[i];
    }
    return sum / arguments.length;
}

avg(2, 3, 4, 5); // 3.5 invoked as a function
avg.call(null, 2, 3, 4, 5); // 3.5  invoked using call (1st parameter specifies this)
```

## Closure

```
function makeFunc() {
    const name = "Mozilla";
    function displayName() {
        console.log(name);
    }
    return displayName;
}

const myFunc = makeFunc();
myFunc();
```

Will this run and what will it do?

What is myFunc?

## What’s happening here?

The string "**Mozilla**" will be displayed in the browser console if you run with Chrome DevTools.

Note that the **displayName()** ***inner function*** is returned from the outer function. You can execute the returned function.

The fact that the code still works may seem **unintuitive**.

***Normally***, the local variables within a function ***only exist*** for the duration of that function's execution.

Once ***makeFunc()*** has finished executing, it is reasonable to expect that the ***name*** variable will no longer be accessible.

## Why does the code still work?

Since the code still works as indicated, this is obviously **not** the case.  Why?

***myFunc*** has become a **closure** — a special kind of **object** that combines two things:

- ★ a ***function***, and
- ★ the ***environment*** in which that function was created

The environment consists of **any local variables** that were **in-scope** at the time that the closure was created.

***myFunc*** is a closure that incorporates both the ***displayName*** function and the ***"Mozilla"*** string that existed when the closure was created.

## TODO

Write a function factory ***convertFromMiles(to)*** that creates and **returns a closure** with a function that converts from **miles** to **another unit**. Your function factory should be able to create functions that convert from either **miles to km** OR **miles to feet**.

The input ***to*** in your function factory invocation can be ***"km"*** OR ***"ft"***

1 mile = 1.60934 km
1 mile = 5280 ft

Create a ***milesToKm*** closure and a ***milesToFt*** closure.

Test in Chrome DevTools.

## TODO

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}
```

***makeAdder*** is a **function factory** — it creates functions that can add a specific value to their argument.

Create an ***addTwo*** closure and an ***addTen*** closure.

They should share the **same function** definition, but **different environments**.

## Template literals

**String interpolation**: New syntax for working with string literals, which can contain embedded expressions

Template literals are surrounded by backtick `   ` symbols

Embed expressions in template literal by

- surrounding them with curly braces and
- prefixing with the dollar sign

## Template literal example

```
const customerName = "John Smith";
console.log(`Hello ${customerName}`);

function getCustomer(){
      return "Allan Lou";
}
console.log(`Hello ${getCustomer()}`);
```

What is the output of this code?

## Multiline string

```
const message = `Please enter a password that
has at least 8 characters and
includes a capital letter`;

console.log(message);
```

Using backticks, you can write multiline strings without needing to concatenate them or use the backslash character

**Spacing is preserved when string is displayed**

## Tagged template strings

Can tag a template string by **preceding it with a function name**.

The template string will be evaluated first, then passed to the function as first argument, for further processing.

String parts of template string are passed to function as array of strings. Blank space between and after expressions are also considered string parts.

All expressions that were evaluated in template string are passed to function as separate arguments.

You can add any number of arguments after the first one

## Tagged template string example

```
function currencyAdjustment(stringParts, region, amount) {
        console.log( stringParts);
        console.log( region );
        console.log( amount );
        let sign;
        if (region==1){
                sign="$";
        }
        else {
                sign='\u20AC'; /* Euro sign */
                amount=0.9*amount; /* currency conversion */
        }
        return `${stringParts[0]}${sign}${amount}${stringParts[2]}`;
}

const amount = 100;
const region = 2;
const msg = currencyAdjustment`You've earned ${region} ${amount}!`;
console.log(msg);
```

## Optional parameters and default values

**ES5 Code:** `state` **as optional parameter**

```
function calcTaxES5(income, state){
        state = state || "Florida";
        console.log("ES5. Calculating tax for the resident of " +
                    state + " with income of $" + income);
}
calcTaxES5(50000);
```

We enter the function body, check whether `state` is provided, if not use `Florida`

In ES6, you can specify default values for function parameters (arguments) that will be used if no value is provided during function invocation.

How?

## Optional parameters and default values

**ES6 Code:** `state` **as optional parameter**

```
function calcTaxES6(income, state = "Florida"){
      console.log(`ES6. Calculating tax for the resident of ${state} with
income of $${income}`);
}
calcTaxES6(50000);
```

How is this different?

## Classes

ES6 introduces classes to JavaScript.

You can now use classes and inheritance in JavaScript -- not just prototypal inheritance

Class declarations are ***not hoisted***, so you need to declare a class before using it.

Note the special function called `constructor`

```
class Pony {
     constructor(color) {
          this.color = color;
     }

     toString() {
          return `${this.color} pony`;
     }
}
const bluePony = new Pony('blue');
console.log(bluePony.toString());
```

## Classes: static members

Classes can have static methods

Static methods can be called only on the class directly

Class member/class variables are not supported as in other languages

However, If you need a class property that's shared by multiple instances of the object, you can create it outside of the class declaration

```
class Pony {
    constructor(color) {
        this.color = color;
    }

    toString() {
        return `${this.color} pony`;
    }

    static defaultSpeed() {
        return 10;
    }
}
console.log(Pony.defaultSpeed());
Pony.count = 1;
console.log(Pony.count);
```

## Classes: getters and setters

Setters and getters **bind functions** to object properties

This is similar to C#

Note that you assign and retrieve the value of `ponyColor` using dot notation, as if it were a declared property of the Pony object.

Notice absence of the `function` keyword

```
class Pony {
	constructor(color) {
		this.color = color;
	}
	get ponyColor() {
		console.log('get color');
		return this.color;
	}
	set ponyColor(newColor) {
		console.log(`set color ${newColor}`);
		this.color = newColor;
	}
}
const bluePony = new Pony('blue');
bluePony.ponyColor = 'red';
console.log(bluePony.ponyColor);
```

## Classes: inheritance

The presence of classes makes it possible to have inheritance

Note the base class and the derived class

Note the use of the `super` keyword.

Note method overriding is often used to replace functionality of a method in the superclass without changing its code.

```
class Horse {
    constructor(speed) { this.speed = speed;}
    maxSpeed() {
        return this.speed + 10;
    }
}
class Colt extends Horse {
    constructor(speed, color) {
        super(speed);
        this.color = color;
    }
    maxSpeed() {
        return super.maxSpeed() + 10;
    }
}
const colt = new Colt(20, 'blue');
console.log(colt.speed);
console.log(colt.maxSpeed());
```

## TODO

```
class Tax{
    constructor(income){
        this.income = income;
     }
     calculateFederalTax(){
         console.log(`Calculating federal tax for income ${this.income}`);
     }
       calcMinTax(){
         console.log("In Tax. Calculating min tax");
          return 123;
     }
}
```

Create an `IndianaTax` class that inherits from `Tax`.  Be sure to add a `stateTaxPercent` property. Override the `calcMinTax` method to call the parent's version and display "`Indiana Tax: adjusting min tax`". Add a `calculateStateTax` method that behaves like `calculateFederalTax`.

## TODO

```
const theTax = new IndianaTax(50000, 6);
theTax.calculateFederalTax();
theTax.calculateStateTax();
theTax.calcMinTax();
```

Use your classes by running the code above

## Challenges for FUN

Much of the recent material we have covered could make for good interview questions.

There are also some very unexpected behaviors you might prefer to learn about now rather than encounter on your own without warning!

https://medium.com/javascript-in-plain-english/5-tricky-javascript-problems-to-check-before-your-next-interview-part-1-60fdecaa59d6

## Resources

★ http://2ality.com/2015/02/es6-scoping.html -- Variables and scoping in ECMAScript 6
★ http://exploringjs.com/es6/ch_variables.html -- Variables and scoping
★ https://medium.com/front-end-developers/es6-variable-scopes-in-loops-with-closure-9cde7a198744 - ES6 variable scopes in loops with closure
★ https://css-tricks.com/implementing-private-variables-in-javascript/ - A elaboration of the past, present and future ways to make private variables for classes
★ https://scotch.io/tutorials/understanding-javascript-closures-a-practical-approach - Some addition closure examples and explanations