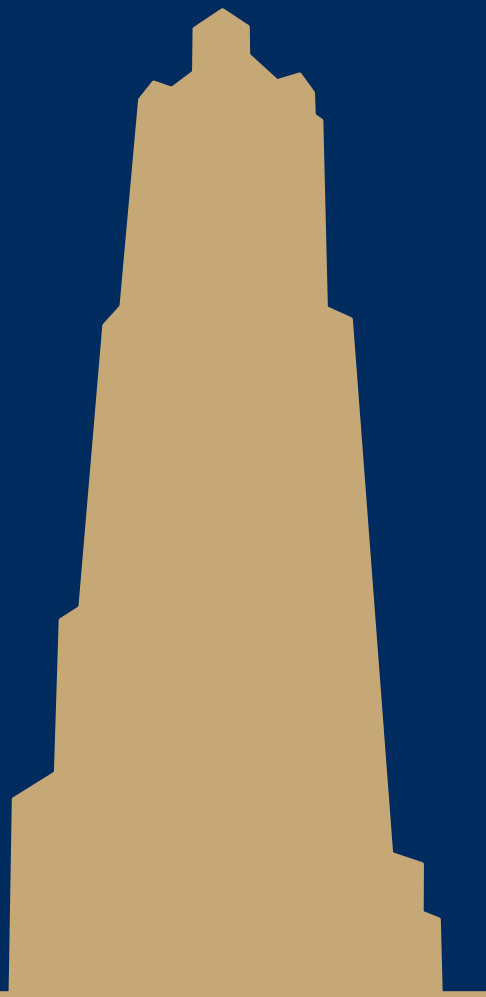
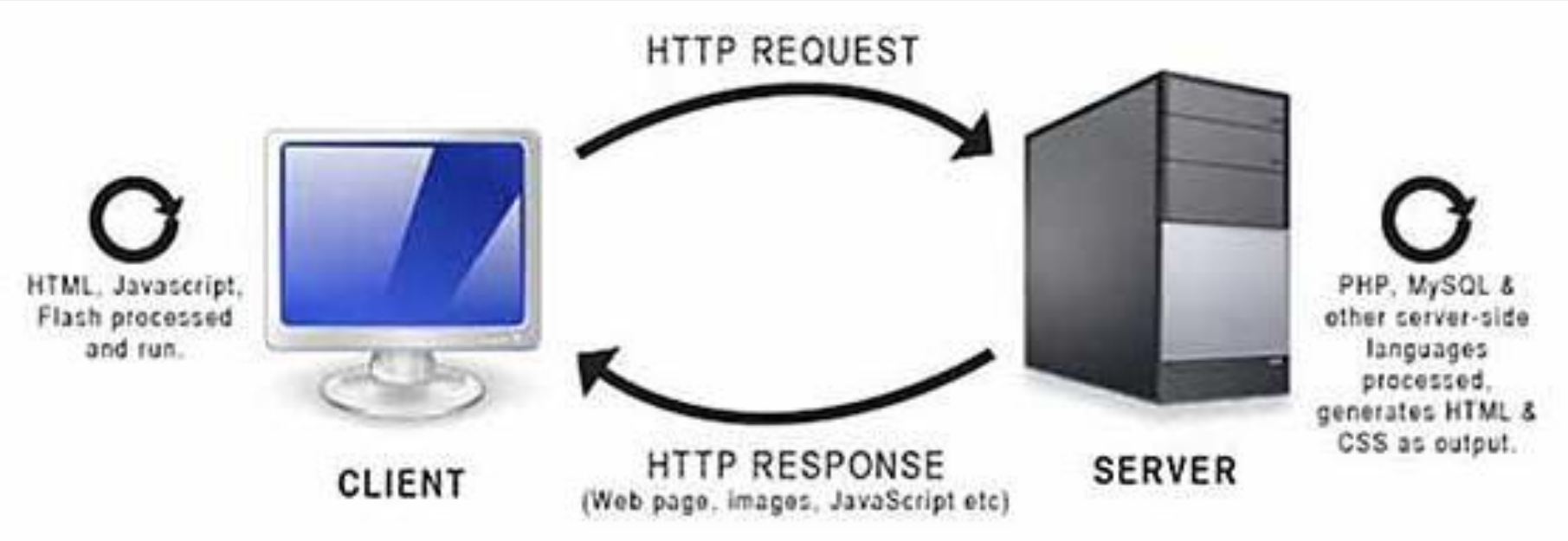
CS/COE 1520

Client-side scripting:
JavaScript and ECMAScript



Client side versus Server Side



The reasoning behind using Scripting Languages in a web page

Why?

- By themselves, HTML and CSS can provide a description of the structure and presentation of a document to the browser
  - A *static* document
- We need to present a *dynamic* application to the user via the browser
  - To do this, we'll need programs that can be fetched from the web and run within the browser
- Examples:
  - The sphere volume webpage
  - The 15 Tile Puzzle webpage

The difference between interpreted and compiled languages

Compiled versus Interpreted languages

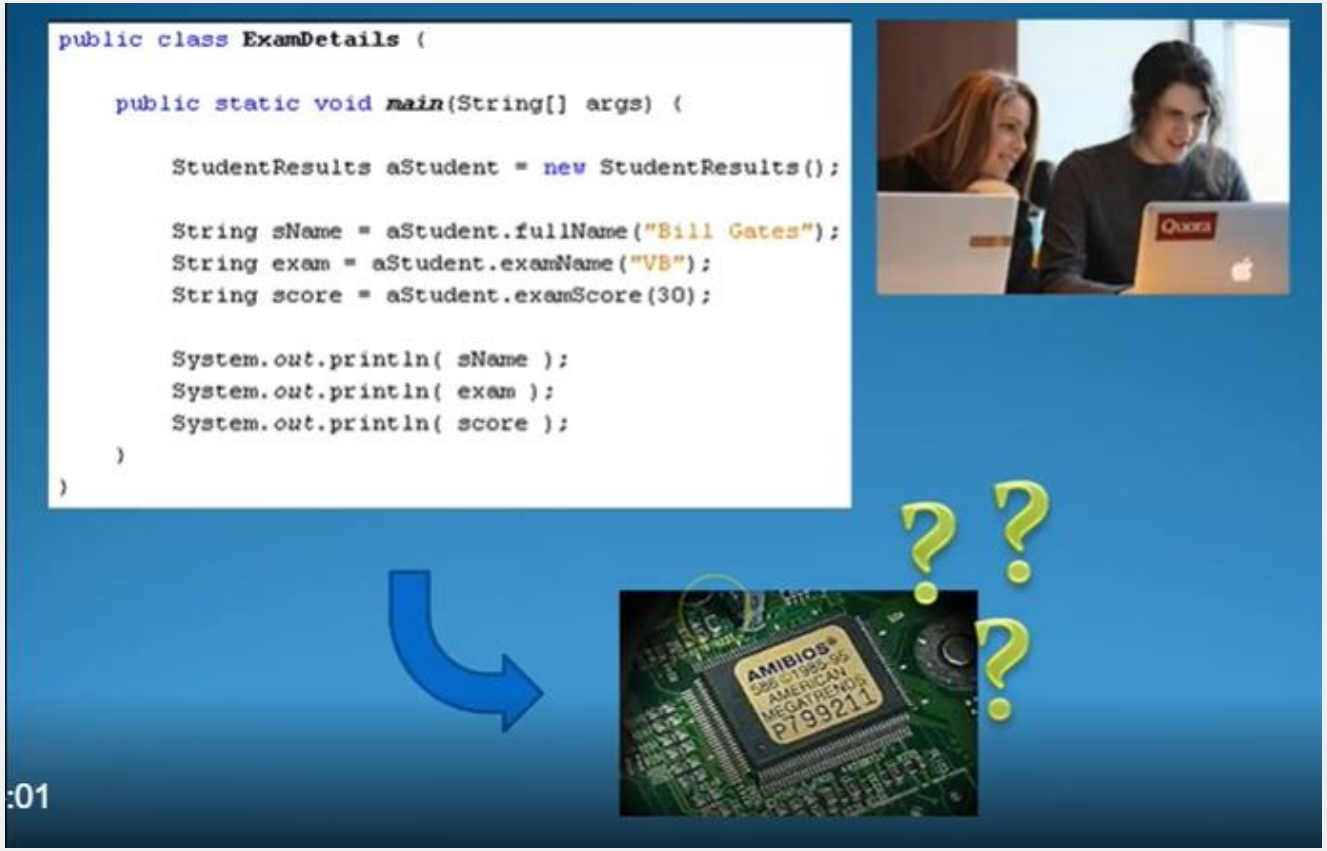
The instructions were compiled in English and the person speaks English (faster)

The instructions were compiled in English and the person speaks Chinese only. She needs an interpreter (slower)

Program is a set of instructions

- Every program is a set of instructions:
  - payValue = hourlyPay * workedTime;
  - data = fetchDataFromServer(“https…”);
  - drivingAge = 16;

Compiled versus Interpreted languages



- Compilers and interpreters take human-readable code and convert it to computer-readable machine code

Interpreted versus Compiled languages

- In a compiled language, the instructions are created (compiled) to a specific machine OS (Windows, Mac OS. This is sent to the client computer.
  - they tend to be faster and more efficient to execute than interpreted languages.
  - You need to use the right compiled file that meets your OS
- In an interpreted language, the source code is not directly translated to a specific machine. Instead, a *different* program, aka the interpreter in the client machine, reads and executes the code.
  - Interpreters reads and executes the program line by
  - They tend to be slower to execute than compiled languages
  - The server can send the “raw” commands does not matter what OS this program is being download into

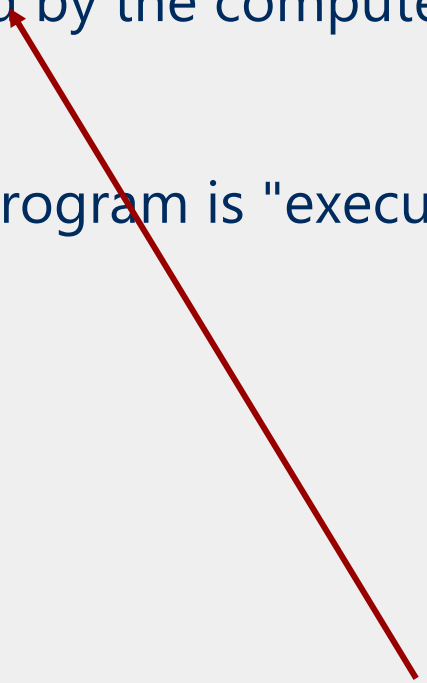
Scripting Languages

Scripting languages

- Programming languages designed for use within a given runtime environment
  - Often to automate tasks for the user
    - E.g.
      - bash, zsh, fish
      - Perl
      - Python
  - These languages are often *interpreted*
    - As opposed to being *compiled*

Compiled vs Interpreted

- Compiled: before being run, a program is compiled into machine code which is executed by the computer
  - E.g., C, C++, C#
- Interpreted: source code of a program is "executed" directly by an interpreter application
  - E.g., Python, Perl, Ruby, PHP
- Pretty simple, right?
  - What about Java?



Java doesn't fit this definition

JavaScript

- JavaScript: The de facto web client-side scripting language
- JavaScript source code can be embedded within or referenced from HTML
  - Through the use of the `<script></script>` tag js01_basic
- It is an interpreted language
  - JavaScript evaluated by the browser in rendering the HTML documents that contain/reference it
  - JavaScript *engines* are the portion of the browser that interpret JavaScript
    - Chrome has V8
    - Firefox has Spidermonkey

JavaScript Basics

JavaScript basics

- Variable names
  - Are case sensitive: `Age` and `age`
  - Cannot be keywords: `for = 2;`
  - Must begin with `$`, `_`, or a letter
    - Followed by any sequence of `$`'s, `_`'s, letters, or digits
- Numeric operators similar to those you know and love:
  - `+`, `-`, `*`, `/`, `%`, `++`, `--`
- Comparison and boolean operators, too:
  - `==`, `!=`, `<`, `>`, `<=`, `>=`, `&&`, `||`, `!`

JavaScript basics

- Strings
  - Have the + operator for concatenation:
    - `fullName = firstName + " " + secondName;`
  - Have `charAt()`, `indexOf()`, `toLowerCase()`, `substring()`, and many more methods
- Control statements similar to Java
  - `if`, `while`, `do`, `for`, `switch`
- Overall, it looks kind of like Java – intentionally

JavaScript is *dynamically typed*

Dynamically Typed variables

- As opposed to the *statically typed*
  - E.g., in Java and C: `int age = 25;`
  - JavaScript: `age = 25;`
- Essentially, types are tied to values, not variables
- The types of the values stored in a given variable is determined at runtime
  - And can change over the run of the program! ➔ `age = "Hello";`
  - This means that checks for type safety are evaluated at run time
  - Js02_vars_types

Type systems

Type Checking:

- the process of verifying and enforcing the constraints of types
- Static Type Checking:
  - The type of a variable is known at compile time instead of at runtime
- Dynamic Type Checking:
  - verifying the type safety of a program at runtime

Type-safe:

- the possibility of type errors is kept to a minimum

Dynamic Type Checking

```
var a = 3;
var b = 4;
if (isNaN(a) || isNaN(b)) {
    console.log(“I cannot add a and b”);
} else {
    console.log(a + b);
}
```

Working with JavaScript's type system

- The `+` operator:
  - If one operand is a string value, the other will be coerced into a string and the two strings will be concatenated
- Numeric operators:
  - If one operand is a string value and it can be coerced to a number (e.g., `"5"`), it will be
  - If string is non-numeric, result is `NaN`
    - (Not A Number)
  - We can also explicitly convert the string to a number using parseInt( ) and parseFloat( )
- Comparisons:
  - `==` and `!=` allow for type coercion
    - What does this mean?

js02_vars_types and js03_more_vars_types

Comparing both type and value

- An additional equality operator and inequality operators are defined to help deal with odd behavior presented by == and !=:
  - === returns true only if the variables have the same value and are of the same type
    - If type coercion is necessary to compare, returns false
  - !== returns true if the operands differ in value or in type

Functions

- `function foo(param1 , param2, param3) { … }`
- Return types are not specified
- Param types are not specified
- Functions execute when they are called, just as in any language
  - Because of this, function definitions should be in the head HTML element
  - E.g., `<head><script>function … </script></head>`

```
function myFunction(a, b) {
  return a * b;   // Function returns the product of a and b
}

let x = myFunction(4, 3);   // Function is called, return
                            // value will end up in x

function addOnlyTwoNumbers(a,b) {
   if (isNaN(a) || isNaN(b))
      return false;
   return Number(a) + Number(b);
}
```

Functions

- Parameters are all passed by value
- No parameter type-checking
- Numbers of formal and actual parameters do not have to correspond
  - Extra actual parameters are ignored
  - Extra formal parameters are undefined
  - All actual parameters can be accessed regardless of formal parameters by using the arguments array

JavaScript has *first-class functions*

- Functions are treated as first-class citizens
  - Meaning they can be:
    - Stored in data structures
    - Assigned to variables
    - Passed as arguments to other functions
    - Returned from other functions

js04_functions

JavaScript arrays

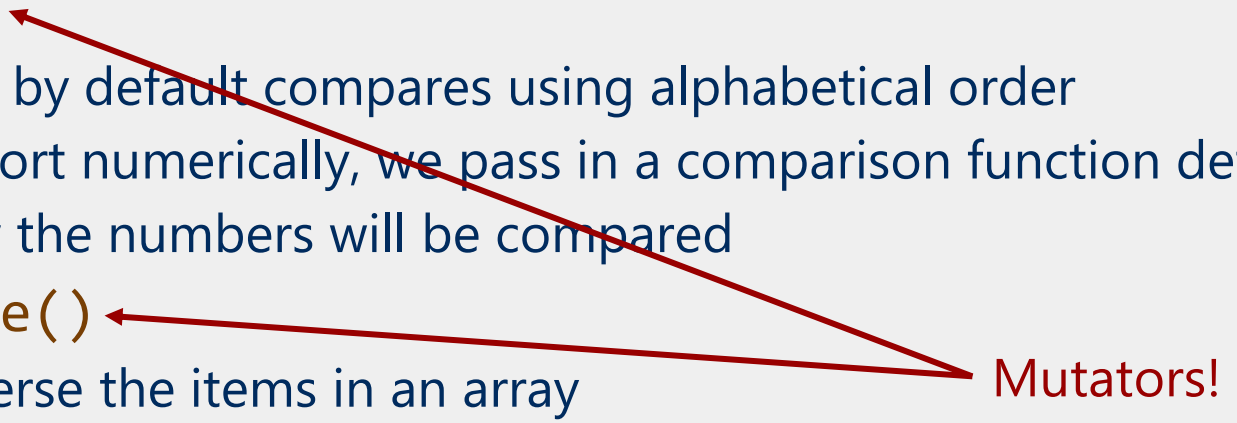
- More relaxed compared to Java arrays
  - Size can be changed and data can be mixed
- Multiple ways to create arrays:
  - Using the new operator and a constructor with multiple arguments:
    - `let A = new Array("hello", 2, "you");`
  - Using the new operator and a constructor with a single numeric argument
    - `let B = new Array(50);`
  - Using square brackets to make a literal
    - `let C = ["we", "can", 50, "mix", 3.5, "types"];`

JavaScript array length

- Like in Java, length is an attribute of all array objects
- In JavaScript it does not necessarily represent the number of items or even memory locations in the array
  - Actual memory allocation is dynamic and occurs when necessary
  - An array with length == 1000 may in fact only have memory allocated for only a 5 elements
- When accessed, empty elements are `undefined`

Some JavaScript array methods

- `concat()`
  - Concatenate two arrays into one
- `join()`
  - Combine array items into a single string (commas between)
- `push()`, `pop()`, `shift()`, `unshift()`
  - Push and pop are a "right stack" (to/from end)
  - Shift and unshift are a "left stack" (to/from beginning)
- `sort()`
  - Sort by default compares using alphabetical order
  - To sort numerically, we pass in a comparison function defining how the numbers will be compared
- `reverse()`
  - Reverse the items in an array



function AddOnlyTwoNumbers(a, b) { if (isNaN(a) || isNaN(b)) return false; return Number(a) + Number(b); }

Sorting comparison function pseudocode

```
function compare(a, b) {
   if (a is less than b by some ordering criterion) {
      return -1;
   }

   if (a is greater than b by the ordering criterion) {
      return 1;
   }

   // a must be equal to b
   return 0;
}
```

js05_arrays

JavaScript objects are not what you're used to

- Not really object-oriented
  - Do not support alot of common features of object-oriented languages, e.g.:
    - Class inheritance
    - Polymorphism
- Are really just an implementation of a map or symbol table

JavaScript objects

- JavaScript objects are represented as property-value pairs
  - Property values can be data or functions
    - Allowing you to basically create methods

```
let my_tv = new Object();
    my_tv.brand = "Samsung";
    my_tv.size = 46;
    my_tv.jacks = new Object();
    my_tv.jacks.input = 5;
    my_tv.jacks.output = 2;
```

Initializing new objects

- Note that the objects can be created and their properties can be changed dynamically
- Objects all have the same type: Object
  - "Constructor" functions for objects can be written, but these do not create new data types, just easy ways of uniformly initializing objects

```
function TV(brand, size, injacks, outjacks) {
    this.brand = brand;
    this.size = size;
    this.jacks = new Object();
    this.jacks.input = injacks;
    this.jacks.output = outjacks;
}
…
let my_tv = new TV("Samsung", 46, 5, 2);
```

js06_objects and js07_more_objects

ECMAScript

These are not the only objects available...

- To talk about other objects, we need to discuss ECMAScript
- What is ECMAScript?
  - ECMA: European Computer Manufacturers Association (ECMA)
  - A standards organization similar to ANSI or ISO
  - ECMAScript is based on JavaScript

What is ECMAScript?

- ECMA standard ECMA-262
- A specification for implementing a scripting language
- Created to standardize the scripting language developed out of Netscape by Brendan Eich
- ECMA-262 tells you how to implement a scripting language
  - JavaScript documentation tells you how to use an implementation of ECMA-262

A little bit of history

- 1995: JavaScript developed and released by NetScape
- 1996: NetScape submits a standard for JavaScript to ECMA
- 1997: 1st edition of ECMAScript published
- 1998: 2nd edition published
- 1999: 3rd edition published
- 2007: Work on 4th edition begins
  - Due to political infighting in the working group, the contributions of the 4th edition are almost completely abandoned
- 2009: 5th edition is published
- 2015: 6th edition (aka ECMAScript 2015) published
- 2016: ECMAScript 2016 (aka ES2016) published
- 2017: ECMAScript 2017 (aka ES2017) published
- …
- ES.Next always refers to the next version in development

ECMAScript Few Features

ECMAScript
Features

Enabling strict mode

- Either:
  - `"use strict";`

    or

  - `'use strict';`
- Appears before any other statement
- If placed before any other statement in a script, the entire script is run using strict mode
- Can also be used to set individual functions to be evaluated in strict mode by placing it before any other statements in a function

Raises errors on variable name typos

- The following will raise a `ReferenceError`:
  - ```
    let myVar = 12;

    mVar = 13;
    ```
    ```

No duplicate function arguments

- ```
  function foo(a, b, a, a) {
      console.log(a);
      console.log(b);
      console.log(a);
      console.log(a);
  }
  foo(1, 2, 3, 4);
  ```
  ```

Paving the way for future ECMAScripts

- The following are treated as reserved words in strict mode:
  - `implements`
  - `interface`
  - `package`
  - `private`
  - `protected`
  - `public`
  - `static`
  - `yield`

strict scripts vs strict functions

- Be very cautious with making a script strict…
  - Consider concatenating two scripts together:
    - sloppy_script + strict_script
      - Result will be sloppily evaluated
      - The `"use strict";` from the strict_script will no longer come before the first statement
    - strict_script + sloppy_script
      - Result will be treated as strict!
      - Could result in errors from strict evaluation of sloppy code!

Arrow functions

- Succinct, anonymous function definitions:

```
hello = function() {
  return "Hello World!";
}

hello = () => {
  return "Hello World!";
}
```

- `myFunction = function foo(a, b, a, a) {…, return xyz}`
- `myFunction = (a) => { return a + 1; }`
- `myFunction = a => a + 1;`
- `myFunction = (a, b, c) => { return a + b + c; };`
- `myFunction = (a, b, c) => { console.log(a); console.log(b); console.log(c); }`

- Very convenient for passing functions as arguments or return values!

Template strings

- Defined with backticks (` not ' or ")
- ```
  `Can span
  multiple lines`
  ```
- ```
  let a = 1;
  let b = 2;
  let s = `Can reference vars like ${a} and ${b}`;
  ```
- ```
  let t = `Can include expressions like ${a + b}`;
  ```
  ```

let and const

- Both alternatives to `var` for variable declaration
- `const` variables cannot be reassigned
  - Note that this does not mean values are immutable…
- `let` allows you to declare variables limited in scope to the block, statement, or expression where they're used
- ```
  var a = 1;
  var b = 2;
  if (a === 1) {
     var a = 11;
     let b = 22;
     console.log(a);  // 11
     console.log(b);  // 22
  }
  console.log(a); // 11
  console.log(b); // 2
  ```
  ```

for … of and iterables

- ES6 instroduces iterators, iterables, and a for loop syntax for iterables
- ```
  let iterable = [10, 20, 30];
  for (let value of iterable) {
      value += 1;
      console.log(value);
  }
  console.log(iterable);
  ```
  ```

for … of vs for … in

- Both are valid in JavaScript
- `for … in` iterates through the enumerable properties of an object in an arbitrary order
- `for … of` iterates over an iterable object
-
```
const iterable = [10, 20, 30];
for (const x in iterable) {
    console.log(x);
}
```
  - Logs: `"0"`, `"1"`, then `"2"`

ES2015 classes

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    display() {
        console.log("Name: " + this.name);
        console.log("Age: " + this.age + "\n");
    }
}
```

ES2015 class notes

- Classes cannot be instantiated before their definition
  - I.e.., class definitions are not "hoisted"
- Class method are not constructable
  - Cannot be used on the right of a new
- Class bodies are evaluated in strict mode
- All instance attributes must be defined in method bodies

Class properties/methods

- Class properties must be set outside of the class body:
  - `Person.species = "Homo sapiens";`
- The `static` keyword can be used to define class methods

Inheritance

```
class Student extends Person {
    constructor(name, age) {
        super(name, age);
        this.classes = [];
    }
    add_class(new_class) {
        this.classes.push(new_class);
    }
    display() {
        super.display()
        console.log("Classes: " + this.classes + "\n");
    }
}
```

Getter/setter methods

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    getArea() {
        return this.calcArea();
    }
    calcArea() {
        return this.height * this.width;
    }
}
const square = new Rectangle(10, 10);
console.log(square.area);  // 100
```

One last ES2015 contribution to highlight

- Tail call optimization
  - ```
    function factorial(n, acc = 1) {
        if (n <= 1) return acc;
        return factorial(n - 1, n * acc);
    }
    ```
    ```