CLASSES IN JAVASCRIPT

WHAT IS A CLASS?

- Classes are a template for creating objects. They encapsulate data with code to work on that data. Classes in JS are built on prototypes but also have some syntax and semantics that are not shared with ES5 class alike semantics.

- Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

CLASS DECLARATIONS

- One way to define a class is using a **class declaration**. To declare a class, you use the class keyword with the name of the class ("Rectangle" here).

```
class Rectangle {
        constructor(height, width) {
                this.height = height;
                this.width = width;
        }
}
```

CLASS EXPRESSIONS

- A **class expression** is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body. (it can be retrieved through the class's (not an instance's) name property, though).

- Un-named Example

```
let Rectangle = class {
        constructor(height, width) {
                this.height = height;
                this.width = width;
        }
};

console.log(Rectangle.name);
output: "Rectangle"
```

- Named Example

```
let Rectangle = class Rectangle2 {
        constructor(height, width) {
                this.height = height;
                this.width = width;
        }
};

console.log(Rectangle.name);
output: "Rectangle2"
```

CLASS BODY AND METHOD DEFINITIONS

- The body of a class is the part that is in curly brackets {}.
- This is where you define class members, such as methods or constructor.

STRICT MODE

- The body of a class is executed in strict mode, i.e., code written here is subject to stricter syntax for increased performance, some otherwise silent errors will be thrown, and certain keywords are reserved for future versions of ECMAScript.

CONSTRUCTOR

- The constructor method is a special method for creating and initializing an object created with a class. There can only be one special method with the name "constructor" in a class. A SyntaxError will be thrown if the class contains more than one occurrence of a constructor method.
- A constructor can use the super keyword to call the constructor of the super class.

PROTOTYPE METHODS

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

```
class Point {
        constructor(x, y) {
                this.x = x;
                this.y = y;
        }
        static distance(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;
        return Math.hypot(dx, dy);
        }
}
const p1 = new Point(5, 5);
const p2 = new Point(10, 10);
p1.distance;   //undefined
p2.distance;   //undefined

console.log(Point.distance(p1, p2)); // 7.071067811865475
```

STATIC METHODS

The static keyword defines a static method for a class. Static methods are called without instantiating their class and **cannot** be called through a class instance. Static methods are often used to create utility functions for an application.

```
class Animal {
        speak() {
                return this;
        }
        static eat() {
                return this;
        }
}
let obj = new Animal();
obj.speak();            // the Animal object
let speak = obj.speak;
speak();                // undefined Animal.
eat()                   // class Animal
let eat = Animal.eat;
eat();                  // undefined
```

BINDING **THIS** WITH PROTOTYPE AND STATIC METHODS

When a static or prototype method is called without a value for this, such as by assigning a variable to the method and then calling it,
the this value will be undefined inside the method. This behavior will be the same even if the "use strict" directive isn't present, because code within the class body's syntactic boundary is always executed in strict mode.

```
function Animal() { }
Animal.prototype.speak = function() {
        return this;
}
 Animal.eat = function() {
        return this;
}
let obj = new Animal();
let speak = obj.speak; speak(); // global object (in
                                non strict mode)
let eat = Animal.eat; eat(); // global object (in
                             non-strict mode)
```

If we rewrite the previous code using traditional function-based syntax in non–strict mode, then this method calls is automatically bound to the initial this value, which by default is the global object. In strict mode, auto binding will not happen; the value of this remains as passed.

INSTANCE PROPERTIES

- Instance properties must be defined inside of class methods:

```
class Rectangle {
        constructor(height, width) {
                this.height = height;
                this.width = width;
        }
}
```

- Static (class-side) data properties and prototype data properties must be defined outside of the Class Body declaration:

```
Rectangle.staticWidth = 20;
Rectangle.prototype.prototypeWidth = 25;
```

FIELD DECLARATIONS

PUBLIC FIELD DECLARATIONS

- With the JavaScript field declaration syntax, the above example can be written as:

```
class Rectangle {
        height = 0;
        width;
        constructor(height, width) {
                this.height = height;
                this.width = width;
        }
}
```

- By declaring fields up-front, class definitions become more self-documenting, and the fields are always present.
- As seen above, the fields can be declared with or without a default value.

PRIVATE FIELD DECLARATIONS

```
class Rectangle {
        #height = 0;
        #width;
        constructor(height, width) {
                this.#height = height;
                this.#width = width;
        }
}
```

- It's an error to reference private fields from outside of the class; they can only be read or written within the class body. By defining things which are not visible outside of the class, you ensure that your classes' users can't depend on internals, which may change version to version.
- Private fields cannot be created later through assigning to them, the way that normal properties can.

SUB CLASSES IN JAVASCRIPT

SUB CLASS DECLARATIONS

- The extends keyword is used in *class declarations* or *class expressions* to create a class as a child of another class.

```
class Animal {
        constructor(name) {
                this.name = name;
        }
        speak() {
        console.log(`${this.name} makes a noise.`);
        }
}
class Dog extends Animal {
        constructor(name) {
                super(name); // call the super class constructor and pass in the name parameter
        }
        speak() {
        console.log(`${this.name} barks.`);
        }
}
let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

- Note that classes cannot extend regular (non-constructible) objects. If you want to inherit from a regular object, you can instead use Object.setPrototypeOf():

```
const Animal = {
        speak() {
                console.log(`${this.name} makes a noise.`);
        }
};
class Dog {
        constructor(name) {
                this.name = name;
        }
}
// If you do not do this you will get a TypeError when you invoke speak
Object.setPrototypeOf(Dog.prototype, Animal);
let d = new Dog('Mitzie'); d.speak(); // Mitzie makes a noise.
```

SPECIES

- You might want to return Array objects in your derived array class MyArray. The species pattern lets you override default constructors.
- For example, when using methods such as map() that returns the default constructor, you want these methods to return a parent Array object, instead of the MyArray object.
  The Symbol.species symbol lets you do this:

```
class MyArray extends Array {
      // Overwrite species to the parent Array constructor
      static get [Symbol.species]() { return Array; }
 }
let a = new MyArray(1,2,3);
let mapped = a.map(x => x * x);
console.log(mapped instanceof MyArray); // false
console.log(mapped instanceof Array); // true
```

MIX-INS

- Abstract subclasses or *mix-ins* are templates for classes. An ECMAScript class can only have a single superclass, so multiple inheritance from tooling classes, for example, is not possible. The functionality must be provided by the superclass.
- A function with a superclass as input and a subclass extending that superclass as output can be used to implement mix-ins in ECMAScript:

```
let calculatorMixin = Base => class extends Base {
        calc() { }
};
let randomizerMixin = Base => class extends Base {
        randomize() { }
};
```

- A class that uses these mix-ins can then be written like this:

```
class Foo { }
class Bar extends calculatorMixin(randomizerMixin(Foo)) { }
```

RE-RUNNING A CLASS DEFINITION

- A class can't be redefined. Attempting to do so produces a SyntaxError.
- If you're experimenting with code in a web browser, such as the Firefox Web Console (**Tools** > **Web Developer** > **Web Console**) and you 'Run' a definition of a class with the same name twice, you'll get a SyntaxError: redeclaration of let *ClassName*;.
- Doing something similar in Chrome Developer Tools gives you a message like Uncaught SyntaxError: Identifier '*ClassName*' has already been declared at <anonymous>:1:1.

REFERENCES

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes