**Lab 2 notes**

**Comparison operators**

Javascript has two sets of comparison operators. The first set are the same as C++, namely == (equality) and != (inequality). The second set has an identity operator === and !==

The identity (===) operator checks both the value *and the* type. In other words, it behaves identically to the equality (==) operator except no type conversion is done, and the types must be the same to be considered equal.

Quick example:

0 == '0' // evaluates to true (types are ignored)

0 === '0' // evaluates to false

false == '0' // true

false === '0' // false

Simple rule of thumb: ALWAYS use the identity operator. NEVER use the equality operator.

Further discussion [here](https://stackoverflow.com/questions/359494/which-equals-operator-vs-should-be-used-in-javascript-comparisons)

**Javascript Arrays**

An empty list or array is declared in Javascript like this:

var fruits = []; // Declare fruits as an empty list.

We can append elements to the array like this:

fruits.push("Oranges"); // added to fruits[0]

fruits.push("Apples"); // added to fruits[1]

fruits.push("Pears"); // added to fruits[2]

We can get the length of an array like this:

var numElems = fruits.length; // returns 3

The simplest way to iterate over an array is to use a for loop with a numeric index. However, there are two other constructs you can use which you can read about [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in#Array_iteration_and_for...in).

To remove an element from an array, use splice(). For example:

// Find and remove item from an array

var index = fruits.indexOf("Apples");

if(index != -1)

{

array.splice(index, 1); // Remove 1 element starting at position index

}

**Javascript Dictionaries**

A dictionary (or hash or map) is a Javascript object that maintains a mapping of key/value pairs. To create an empty dictionary, we just define a generic Javascript object:

var fruits = {};

To add key/value pairs to the dictionary:

fruits["Apples"] = 3; // Key is “Apples”, value is 3

fruits["Oranges"] = 10; // Key is “Oranges”, value is 10

fruits["Pears"] = 5;

In Javascript object notation, fruits looks like this:

{Apples: 3, Oranges: 10, Pears: 5}

Note carefully that properties order in objects is not guaranteed in JavaScript, i.e. a dictionary **is an unordered collection of properties** (keys) each of which contains a primitive value, object, or function. If you want property order, use an array.

To iterate over the dictionary:

for (var fruitName in fruits)

{

if (fruits.hasOwnProperty(fruitName))

{

var quantity = fruits[fruitName];

console.log(quantity);

}

}

Note: The if condition above is necessary only if you want to iterate over the properties which are dictionary object's very own. This is because for..in will iterate through all the inherited enumerable properties.

We can also define a dictionary like this:

// Empty object literal with properties added afterward

var fruits = {};

fruits.Apples = 10;

fruits.Oranges = 5;

// etc.

Or like this:

var fruits =

{

fruits.Apples = 10;

fruits.Oranges = 5;

// etc.

};

**If your properties (keys) are dynamic, you must use the square bracket operator, e.g.**

var fruits = {};

var fruitName = "Pears"; // Here, the key (fruitName) is dynamic

fruits[fruitName] = 10;

Finally, note that keys are always strings.

**Javascript Classes**

Javascript classes are **nothing like** classes in C++ or other similar languages.

In C++, classes are templates for objects. When we want a new object, we *instantiate* the class which means the methods and properties of the class are copied into the new instance. This instance is the object and after instantiation, it has absolutely no active relation with the parent class.

In Javascript, strictly speaking there are no classes - while someone may refer to a Javascript class, they are in fact talking about a constructor. This is because Javascript is not a class- based language (like Java or C++), rather it is a *prototype-based* language. There is a keyword ‘class’ in Javascript, but this is [syntactic sugar](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) (meaning something that is designed to make things easier to read). For now, let’s consider that instantiating a Javascript ‘class’ does create a new object, but it is not independent of its parent class. It creates an object that is linked to a *prototype*. Changes to that prototype propagate to the new object, *even after* instantiation. We will see more on this later, for now let’s focus on how to create classes and objects.

**Creating objects with constructor calls**

Javascript uses **constructor functions** to define objects and their features. Calling *any* function with the new keyword causes it to return an object. This is called making a *constructor call*, and such functions are generally called *constructors*. Let’s see an example:

// Defines a constructor function - note absence of a class definition.

function Rectangle(width, height)

{

this.width = width; // Stores parameter width as an object property.

this.height = height; // Likewise for height.

}

// Defines a method called ‘area’ for the Rectangle object

Rectangle.prototype.area = function()

{

return this.height \* this.width;

}

// Usage..creates a Rectangle object using new, then prints it’s area.

const rectObject = new Rectangle(10, 10);

console.log(rectObject.area());

Note that width and height are analogous to public member variables in C++, i.e. we can do this:

console.log(squareObject.width);

If you want to hide width and height (analogous to private) then we rewrite the above as:

function Rectangle(width, height)

{

var width = width;

var height = height;

this.getWidth = function()

{

return width;

};

this.getHeight = function()

{

return height;

}

}

Rectangle.prototype.area = function()

{

return this.getWidth() \* this.getHeight();

}

Let’s look again at how we create a Rectangle object:

const rectObject = new Rectangle(10, 10);

This looks straightforward. When you call a function with new, four things happen under the hood:

1. A new object gets created (rectObject);
2. rectObject gets linked to another [object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype), called its *prototype*;
3. The function's this value is set to refer to rectObject;
4. The function implicitly returns rectObject.

**Understanding the Prototype chain**

Let’s consider another code example:

function Rectangle(width, height) {  
 this.width = width;  
 this.height = height;  
 this.area = function () {  
 return this.width \* this.height;  
 };  
}

Here, we are defining a method area() and binding it to the this keyword. Note carefully that you are providing that method to only that particular instance and it does not really have any relationship with an object instance of that constructor, pretty much like a static method in C++.

Further, any method attached via this will get re-declared for every new instance we create, which could affect the memory usage of the application negatively if we wish to create so many instances (again, to use the C++ analogy, it’s like having another static method added everytime you create a new instance). Keep in mind the above approach can be useful if you need to access local private variables, i.e.:

function Rectangle(width, height) {  
 var width = width;  
 var height = height;  
 this.area = function () {  
 return width \* height;  
 };  
}

This approach is fine if you are creating a small number of instances and accessing local object variables is part of the code design.

For other scenarios (when the creation of many object instances is expected), the optimal way to define methods for all instances of a class, is to use the object’s prototype.

Almost all objects in Javascript have the *prototype* property and we say its prototype object (or attribute) is Object.prototype. By using this property and the *prototype chain* we can *mimic* inheritance.

**The prototype is a reference to another object and it is used whenever JS can’t find the property you’re looking for on the current object**

*[For the purposes of this discussion, the term property also includes method names].*

If we call a property on an object that does not exist, JavaScript will go to the prototype object and look for it there. If it finds it it will use it, if not it will go to that object’s property and look there. This can bubble up all the way to Object.prototype before returning undefined.

Let’s take another example:

function Shape(name)

{

this.name = name;

}

Shape.prototype.sayName = function()

{

console.log('I am a :' + this.name);

}

function Rectangle(width, height, name)

{

var width = width;

var height = height;

this.name = name;

this.getWidth = function()

{

return width;

};

this.getHeight = function()

{

return height;

}

}

Rectangle.prototype.area = function()

{

return this.getWidth() \* this.getHeight();

}

**// Create a reference for the prototype**

Rectangle.prototype = Object.create(new Shape());

var rect = new Rectangle(10,10,'Simple Rectangle');

console.log('Rectangle area: ' + rect.area());

rect.sayName();

rect.doesNotExist();

Here, the method area() is applied to the prototype of the object, so it is only stored in memory once. When we call area() it is executed in the context of the current object (rect) and we get the correct output.

When we call sayName(), it doesn’t exist on Rectangle, so it goes up the prototype chain to Shape, where it is found and executed.

Finally, when we call doesNotExist(), the search goes all the way up the prototype chain to Object.prototype where it isn’t found, causing an error to be thrown.

Think of it as working this way - we request a property on an object that it does not have and the object says “I don’t have this property, ask my prototype”. The process of referring lookups for non-existing properties is called *delegation*. This does pose a small overhead but it is not significant unless you have a deep prototype hierarchy.

Before the JavaScript engine executes a program, it builds an environment to run it in, in which it creates a function, called [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object) (i.e. the constructor function for class Object), and an associated object, called [Object.prototype](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype).

In other words, Object and Object.prototype *always* exist, in *any* executing JavaScript program. In simple terms, Object.prototype allows the addition of properties to all objects of type Object. We will see some uses of Object now.

Object’s can be interrogated to see if they have a specific property, e.g:

console.log(rect.hasOwnProperty("name")); // true

We can also use Object to interrogate an object for it’s properties this way:

console.log(Object.hasOwnProperty(rect, toString)); // false

However, note carefully that:

console.log(rect.toString());

Does not give a ReferenceError as you might expect, rather it outputs [object Object].

(which basically means rect is an object that contains some properties).

This is because while rect does not have a toString property, *the prototype* of rect does.

In this example, we delegated the lookup to the prototype.

We can access non-existent properties on an object as long as (that object’s) prototype has those properties. We can assign properties and methods to an object’s prototype so that we can use them as if they existed on the object itself.

If several objects share the same prototype, they can all access that prototype’s properties, immediately after we assign them, without having to copy those properties or methods to each individual object. Let’s see a quick example where we override the default Object.prototype.toString method for our Rectangle class:

Rectangle.prototype.toString = function() {  
 return this.name + " width: " + this.getWidth() + " height: " +

this.getHeight();  
};

Then:

var rect = new Rectangle(10,10,'Simple Rectangle');

// prints 'Simple Rectangle' instead of [object Object]

console.log(rect.toString());

Note, you must comment out this line:

Rectangle.prototype = Object.create(new Shape());

Otherwise, the object prototype for Rectangle is now set to Shape which does not have a toString() method, so the call will be delegated to Object.prototype.

To summarise:

* Objects do *not* "inherit" a toString property;
* Objects do *not* "inherit" from Object.prototype *at all*;
* All objects of the same class *are* **linked** to Object.prototype;
* All objects of the same class are linked to the *same* Object.prototype.
* To find the prototype of an object, you use: Object.getPrototypeOf(objectName).

Objects do not "inherit from" their prototypes - they *delegate* to them.

As a final note, Object.hasOwnProperty() only checks properties directly on the object, not anywhere in the prototype chain. For example, returning to our Rectangle example:

function Rectangle(width, height)

{

this.width = width; // Stores parameter width as an object property.

this.height = height; // Likewise for height.

}

// Defines a method called ‘area’ for the Rectangle object

Rectangle.prototype.area = function()

{

return this.height \* this.width;

}

console.log( rectObject1.hasOwnProperty("area")); // prints false

console.log( rectObject1.hasOwnProperty("width")); // prints true

References and Further reading:

https://hackernoon.com/understanding-javascript-prototype-and-inheritance-d55a9a23bde2

https://www.thecodeship.com/web-development/methods-within-constructor-vs-prototype-in-javascript/

<https://scotch.io/tutorials/better-javascript-with-es6-pt-ii-a-deep-dive-into-classes>

<https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes>

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes>

<https://stackoverflow.com/questions/4691044/should-i-use-prototype-or-not>