Kenny R N Wilson

JavaScript

Cheat Sheets

LINQ

Name	Code	Result	Description
every	`[1,3,5].every(x=> x%2 == 1)`	`true`	
some	`[1,2].some(x=> x%2 == 1)`	`true`	
filter	`[1,2,3,5].filter(x=> x%2 == 1)`	`[1,3,5]`	
find	`[1,2].find(x=> x%2 == 0)`	`2`	
findIndex	`[1,2].findIndex(x=> x%2 == 0)`	`1`	
map	`[1,2].map(x=>x*2`	`[2,4]`	
reduce	`[1,2].reduce((acc,curr)=>acc+curr,0)`	`3`	*Also know as inject or fold*
reduceRight	`[2,3].reduce((acc,curr)=> Math.pow(acc,curr))`	`8`	*Work right to left*
Flat	`[1,2,[3,4],[5,6]].flat()`	`[1,2,3,4,5,6]`	*Flatten any elements that are arrays*

Arrays as Stacks and Queues

Name	Code	Result	Description

pop	`let a = [1,2,3];`
	`console.log(a.pop());`
	`console.log(a);`
		`3`
		`[1,2]`
			Remove the last element and returns it

push	`let a = [1,2,3];`
	`console.log(a.push(4));`
	`console.log(a);`
		`4`
		`[1,2,3,4]`
			Append to end of array

shift	`let a = [1,2,3];`
	`console.log(a.shift());`
	`console.log(a);`
		`1`
		`[2,3]`
			Remove and return the first element

unshift	`let a = [1,2,3];`
	`console.log(a.unshift(4));`
	`console.log(a);`
		`4`
		`[4,1,2,3]`
			Append to front of array

Array Slice

Slice does not modify the original array.

Name	Code	Result	Description
slice	`[1,2,3,4,5].slice(2,4)`	`[3,4]`	
slice	`[1,2,3,4,5].slice(2)`	`[3,4,5]`	
slice	`[1,2,3,4,5].slice(-1)`	`[5]`	
slice	`[1,2,3,4,5].slice(-3,-1)`	`[3,4]`	

Array Splice

Splice modifies the array in place. In inserts and removes

Name	Code	Result	Description

splice	`a = [1,2,3,4,5]`
	`a.splice(1)`
	`a`
		`[2,3,4,5]`
		`[1]`
			Removes and returns all elements starting at index 1.

splice	`a = [1,2,3,4,5]`
	`a.splice(1,2)`
	`a`
		`[2,3]`
		`[1,4,5]`
			Removes and returns 2 elements starting at index 1.

splice	`a = [1,2,3,4,5]`
	`a.splice(1,2,8,9)`
	`a`
		`[2,3]`
		`[1,8,9,4,5]`
			Replace two elements starting at index 1 with 8 and 9

splice	`a = [1,2,3,4,5]`
	`a.splice(1,2,8,9,10,11)`
	`a`
		`[2,3]`
		`[1,8,9,10,11,4,5]`
			Replace two elements starting at index 1 and add four elements 8,9,10,11

Array Fill

Name	Code	Result	Description

fill	`a = new Array(3)`
	`a.fill(3)`
	`a`
		`<3 emppty items>`
		`[3,3,3]`
		`[3,3,3]`
			Fill the array with the value 3

fill	`a = new Array(3)`
	`a.fill(3,1)`
	`a`
		`<3 emppty items>`
		`[undefined,3,3]`
		`[undefined,3,3]`
			Fill the array with the value 3 starting at index 1

Fill	`a = new Array(4)`
	`a.fill(3,1,2)`
	`a`
		`<4 emppty items>`
		`[undefined,3,3,undefined]`
		`[undefined,3,3,undefined]`
			File the array starting and index 1 and ending at 2

Copy Within

Highly performance and modelled on memmove from C.

Name	Code	Result	Description
copyWithin	`a = [1,2,3,4,5]` `a.copyWithin(1)`	`[1,1,2,3,4,5]`	*Copy array from element 0 to array starting at index 1.*
copyWithin	`a = [1,2,3,4,5]` `a.copyWithin(1,3,4)`	`[1,4,5,4,5]`	*Copy array slice at between 3 and 4 to position 1.*

Array Misc.

Name	Code	Result	Description
sort	`a = [5,4,3,2,1]` `a.sort()`	`[1,2,3,4,5]`	*Sort array*
reverse	`a = [5,4,3,2,1]` `a.reverse()`	`[1,2,3,4,5]`	
join	`a = [1,2,3,4,5]` `a.join()`	`"1,2,3,4,5"`	

Functions

Name	Code
Declaration	`function add(a,b) { return a+b}`
Named Expressions	`var f = function add(a,b) { return a+b};`
Anonymous Expression	`var f = function(a,b) { return a+b};`

Rest Parameter	`function avg(...args) {`
	`  var sum = 0;`
	`  for (let value of args) {`
	`  sum += value;`
	`  }`
	`  return sum / args.length;`
	`}`

Default param value	`function add( x, y = 3.0 ) { return x+y};`
Lambda (no parameters)	`var f =  () => 3.0`
Lambda (one parameter)	`var f = x => x * x ;`
Lambda two parameters	`Var x = (x,y) => x + y;`
Spread into Rest parameter	`var numbers = [10,20,30]` `console.log(avg(...numbers));`
Spread into normal parameters list	`function add(x,y) {return x+y;}` `console.log(add(...[3,5]));`

Method	`var o5 = {`
	`  Print: function() {`
	`  console.log("Hello World");`
	`  }`
	`}`

Method shorthand	`var o5 = {`
	`  Print () {`
	`  console.log("Hello World");`
	`  }`
	`}`

Iterators

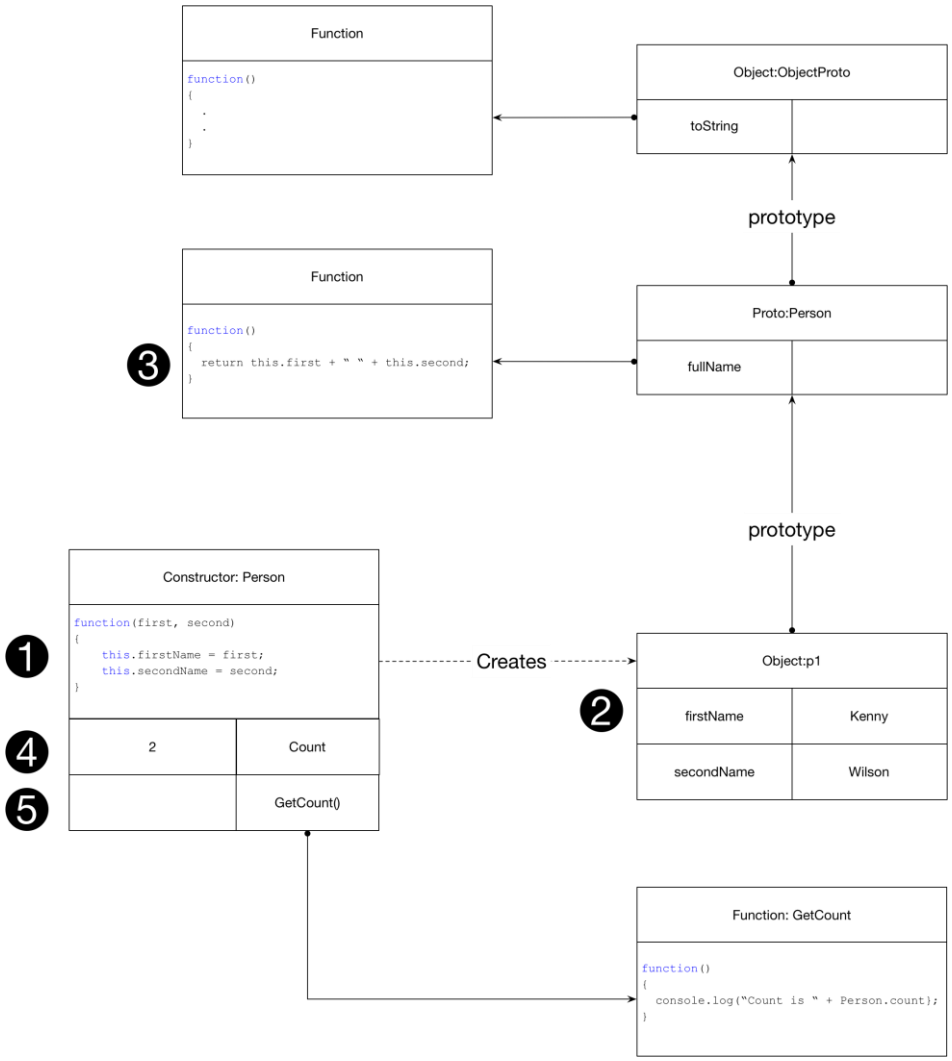
Name	Code

Generate Iterator	`let Iterable = function(count) {`
	`  this.count = count;`
	
	`  this[Symbol.iterator] = function* () {`
	
	`  while (this.count >0)`
	`  {`
	`  yield this.count;`
	`  this.count = this.count-1;`
	`  }`
	`  }`

Consume Iterable with Spread	`console.log(...new Iterable(3));`
Consume Iterable with for/of	`for(let a of [1,2,3])` `  console.log(a*2);`

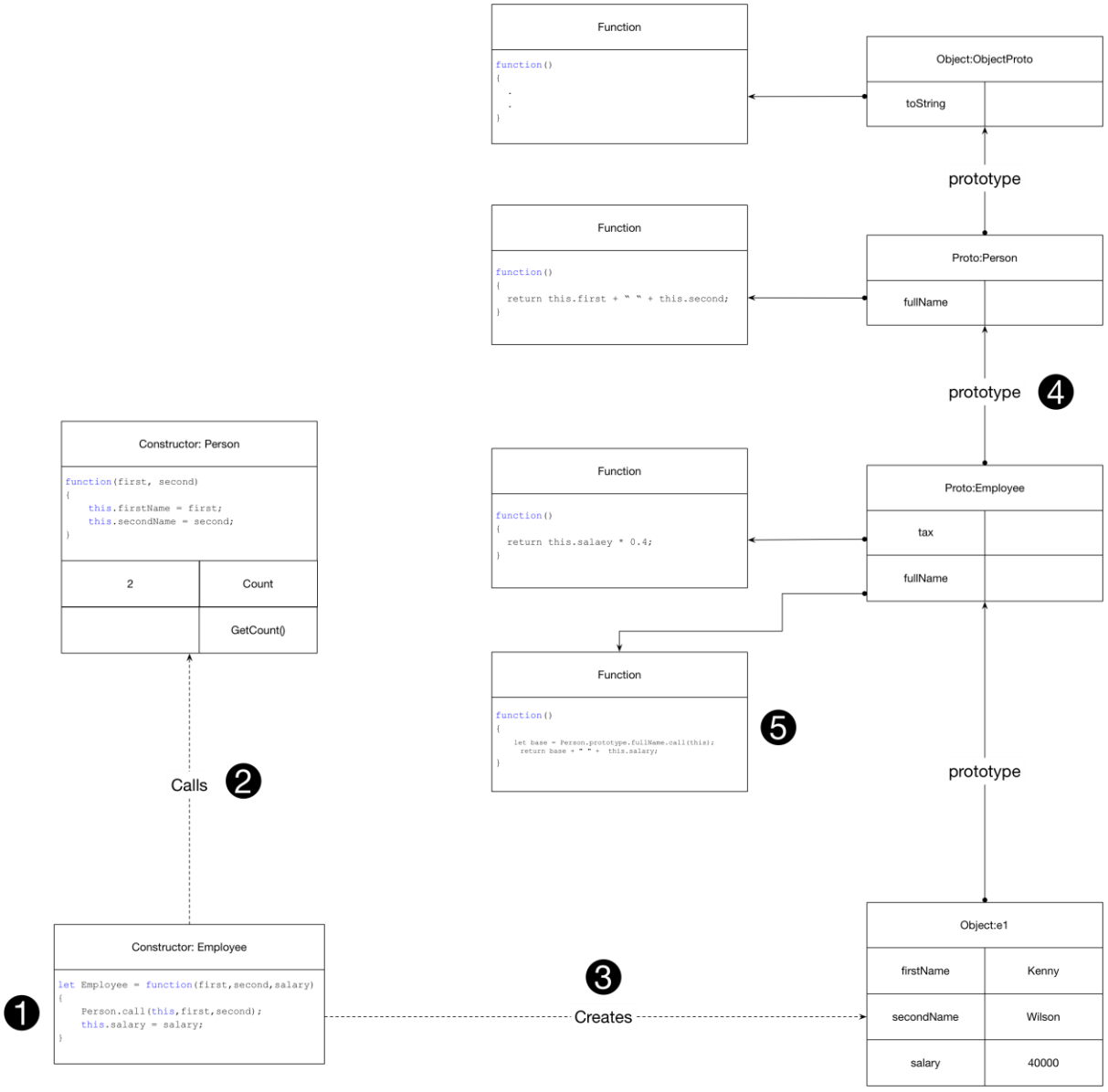
Objects

```
let Person ❶ = function(first, second)
{
    // Instance fields
    this.firstName = first;
    this.secondName = second;
    Person.Count = Person.Count+1;
}

Person.prototype.fullName ❸ = function() {
    return this.firstName + " " + this.secondName;
}

Person.Count ❹ = 0;

Person.GetCount ❺ = function() {
    console.log("Person count is " + Person.Count );
}

let p1 ❷ = new Person("Kenny", "Wilson");
```



Inheritance

```
let Person = function(first, second)
{
    // Instance fields
    this.firstName = first;
    this.secondName = second;
    Person.Count = Person.Count+1;
}

Person.prototype.fullName = function() {
    return this.firstName + " " + this.secondName;
}

Person.Count = 0;

Person.GetCount = function() {
    console.log("Person count is " + Person.Count );
}

let p1 = new Person("Kenny", "Wilson");

Person.GetCount();

let Employee ❶ = function(first,second,salary)
{
    ❷ Person.call(this,first,second);
    this.salary = salary;
}

❹ Object.setPrototypeOf(Employee.prototype,Person.prototype);

Employee.prototype.tax = function () {return this.salary * 0.4};

Employee.prototype.fullName = function() {
    let base = Person.prototype.fullName.call(this);
    return base + " " +  this.salary;
}

let e1 ❸ = new Employee("Kenny", "Wilson", 40000);
```


Function
function()
{
.
.
}
Object:ObjectProto
toString
prototype
Function
function()
{
return this.first + " " + this.second;
}
Proto:Person
fullName
prototype
4
Constructor: Person
function(first, second)
{
this.firstName = first;
this.secondName = second;
}
2
Count
GetCount()
Function
function()
{
return this.salaey * 0.4;
}
Proto:Employee
tax
fullName
Function
function()
{
let base = Person.prototype.fullName.call(this);
return base + " " + this.salary;
}
5
Calls
2
prototype
Constructor: Employee
let Employee = function(first,second,salary)
{
Person.call(this,first,second);
this.salary = salary;
}
1
3
Creates
Object:e1
firstName
Kenny
secondName
Wilson
salary
40000

Objects Prototypes

```
// Constructor defines prototype
function Person(first,second)
{
    // 2 Instance fields
    this.firstName = first;
    this.secondName = second;

    // Increment static field defined below
    Person.numPeople++;
}

// Instance Method
Person.prototype.fullName = function()
{
    return this.firstName + " " + this.secondName;
}

// Instance Property
Object.defineProperty(Person.prototype,"age", {
    get: function() {return this._age;},
    set: function(age) {this._age=age;}
});

// Static field
Person.numPeople = 0;

// Static method
Person.printCount = function()
{
    console.log(Person.numPeople + " people")
}

// Create instance and set age property
var p1 = new Person("Kenny", "Wilson");
p1.age = 20;

// Create second instance
var p2 = new Person("Isla", "Wilson");

console.log(p1.fullName());
console.log(p1.age);
Person.printCount();
```

Classes

```
class Person
{
    // Static Field
    static count = 0;

    // Static Method
    static printPeeople(...people) {
        people.forEach(p => console.log(p.toString()));
    }

    // Constructor
    constructor(first, second) {
        // Initialize Instance Fields
        this.firstName = first;
        this.secondName = second;

        // Increment Static Field
        Person.count++;
    }

    // Instance Method
    toString() {
        return this.firstName + " " + this.secondName + " " + this._age;
    }

    // Instance Getter/Setter
    get age() { return this._age };
    set age(age) { this._age = age };

}

// Subclass
class Employee extends Person {

    constructor(first, second, salary) {
        super(first, second);
        this.salary = salary;
    }

    // Overriden Method
    toString() {
        return super.toString() + ", salary=" + this.salary;
    }
}

var p1 = new Person("Kenny", "Wilson");
p1.age = 45;

var p2 = new Person("Isla", "Wilson");
p2.age = 50;

var e1 = new Employee("John", "Smith", 50000);
e1._age = 99;

Person.printPeeople(p1, p2, e1);
```

Name	Code

Example	`class Person`
	`{`
	`  constructor(first, second)`
	`  {`
	`  this.firstName = first;`
	`  this.secondName = second;`
	`  }`
	
	`  fullName() {`
	`  return this.firstName + " " + this.secondName;`
	`  }`
	`}`

Inheritance	`class Employee extends Person {`
	
	`  constructor(first, second,salary)`
	`  {`
	`  super(first,second);`
	`  this.salary = salary;`
	`  }`
	
	`  fullName() {`
	`  return super.fullName() + " " + this.salary;`
	`  }`
	`}`

Static Method	`class Employee extends Person {`
	
	`  constructor(first, second,salary)`
	`  {`
	`  super(first,second);`
	`  this.salary = salary;`
	`  }`
	
	`  fullName() {`
	`  return super.fullName() + " " + this.salary;`
	`  }`
	
	`  static printAll(...employees) {`
	`  employees.forEach(e=> console.log(e.fullName()));`
	`  }`
	`}`

For in

Loop over elements in an array

```
let a = [1,2,3]
for (const e of a ) { console.log(e);}
```

APIs

- String – MDN
- Number - MDN
- Array - MDN

JavaScript and the DOM

The query selector methods take any valid CSS selector including pseudo selector. It is immensely powerful.

ACCESSING THE DOM

Function	Description
`element.querySelector("h1")`	*Get the first descendent of this element whose tag is "h1"*
`element.querySelector("#myId")`	*Get the first descendent of this element whose id is "myId"*
`element.querySelector(".myClass")`	*Get the first descendent of this element whose class is "myClass"*
`element.querySelectorAll("h1")`	*Get the list of descendent nodes whose tag is "h1"*
`element.querySelector("#myId")`	*Get a node list whose single element is the element with id "myId" or an empty list if no element has such an Id.*
`element.querySelector(".myClass")`	*Get the list of descendent nodes whose class is "myClass"*

`querySelectorAll` returns a non-live list. Any elements that are added or removed are not reflected in the node list returned from this method. If we use the `getElementsByTagName` or `getElementsByClassName` these return live lists. If add or remove elements to the DOM they will be reflected in these lists.

TRAVERSING THE DOM

Function	Description
`element.children`	*Get the child element nodes. Text nodes are excluded*
`element.childNodes`	*Get the child nodes including text nodes.*
`element.firstChild`	*Get the first node*
`element.firstElementChild`	*Get the first element child*
`Element.parentElement`	*Returns the parent element*
`Element.nextSibling`	*Get the next sibling node*
`Element.nextElementSibling`	*Get the next sibling element node*
`Element.previousSibling`	*Get the previous Sibling*
`Element.previousElementSibling`	*Get the previous element sibling*

MODIFYING THE DOM

Function	Description
`element.textContent = "Hello"`	*Set the text content of a node*
`element.className = "class1 class2"`	*Set the class string on the element. We add two classes*
`element.style.backgroundColor = "red"`	*Update the elements inline style*
`Element.innerHtml = "<h1>Different<h1>"`	*Replace entire html content with new content*
`Element.insertAdjacentHtml('beforeend','<h1>Mode</h1>'`	*Add new html just after its last child*
`Element.insertAdjacentHtml('afterbegin,'<h1>Mode</h1>'`	*Add new html just before its first child*
`Element.insertAdjacentHtml('beforebegin,'<h1>Mode</h1>'`	*Before current html*
`Element.insertAdjacentHtml('afterend,'<h1>Mode</h1>'`	*After current html*
`Document.createElement`	*Create a new element*
`Element.appendChild(element)`	*Append newly created element*

`Element.append`	*More flexible than appendChild but not supported by IE*
`Element.prependChild(element)`	*Prepend. Not supported by EI*
`Element.cloneNode`	*Clone a node. Argument specified if clone is deep*
`List.remove`	*Remove child. Not supported is*

If we want to add a existing element somewhere else in the DOM it will be moved and not cloned.

STYLING THE DOM

Function	Description
`element.style.backgroundColor = "red"`	*Update the elements inline style*
`element.className = "class1 class2"`	*Set the class string on the element. We add two classes*
`Element.classList.add("class")`	*Add single class to the className*
`Element.classList.remove("class")`	*Remove single class from className*
`Element.classList.toggle("class")`	*Toggle the class name*

Characteristics and Benefits

- Automatic Garbage Collection
- Immutable Strings
- Untyped Variables

Language Basics

Scope

Scope defines the visibility of identifiers. Scopes can be nested. An identifier can only be accessed from the scope in which it is declared or by any scope nested inside the scope in which it is declared. Identifiers are created when the scope they belong to comes into existence.

JavaScript provides the var, let and const modifiers to specify the scope of variables. Variables declared with var have function or global scope whereas variables declared with let or const have block scope. We will look at let and const first as they are the most modern modifiers.

LET

Block scope

Variables defined with let have block scope and as such are only visible within their enclosing code block.

```
{
    let mylex = 4;
}

console.log(mylex);

>> ReferenceError: mylex is not defined
```

Do not impact global object.

Unlike var, variables declared with let at global scope do not create properties on the global object. If we run the following **in a browser**.

```
var a = 10;
let b = 5;

console.log(this.a);
console.log(this.b);

>> 10
>> undefined
```

Temporal Dead Zone

A variable declared with let comes into existence at the same time as its scope. It cannot, however, be accessed until it is initialized. In contrast with var, the compiler does not provide an automatic initialization to the value undefined at the point a let variable comes into existence.

Consider the following fragment of code. If the shadowing variable with block scope was not in existence when we log, we would see the value 10 from the var defined variable. The fact we see the reference error is proof that on the first line in the block the global variable is already shadowed by the let variable but since it has no initial undefined value so we cannot access it.

```
var a = 10;

{
    console.log(a);

    let a = 5;
}

>> ReferenceError: Cannot access 'a' before initialization
```

This property of let variables leads to an effect known as the temporal dead zone.

```
function f()
{
    console.log(a);
    console.log(b);

    var a =5;
    let b = 6;
}

f();

>> Undefined
>> ReferenceError: Cannot access 'b' before initialization

>> undefined
>> 7
```

Const

Const is like let but the variable must be initialized when it is declared, after which it cannot be re-assigned. In most other behaviours it is the same as let. This includes temporal dead zone.

VAR

A variable declared with var takes the scope of the function it is defined inside or if it is not defined inside a function it takes the global scope.

```
// Global scope
var a = 5;

function f()
{
    // Scoped to the function b
    var b = 15;
}

f();

console.log(a)
console.log(b);

>> 5
>> Uncaught ReferenceError: b is not defined
```

Hoisting

A variable is declared with the var comes into existence at the same time as the scope in which it is declared (function or global). In addition, and in contrast to let, it is also automatically initialized to `undefined` at the point it comes into existence. There is no temporal dead zone with var variables. This is sometimes known as hoisting. It is as if the declaration was hoisted to the top of the scope and initialized as `undefined`.

```
function f()
{
    console.log(a);
    var a;
}
f();

>> undefined
```

If we provide an explicit initial value this takes effect from the point in the code where we make the assignment. The explicit initial value is not hoisted.

```
function f()
{
    console.log(a);
    var a = 5;
    console.log(a);
}
f();

>> undefined
>> 5
```

Blocks have no effect on var.

Blocks have no impact on the scope of variables declared with var.

```
{
    var a = 5;
}

console.log(a);

>> 5
```

Redeclaring

Redeclaring a variable declared initially with var has no effect and does not clear its value.

```
{
    var a = 5;
}

var a;

console.log(a);

>> 5
```

Undeclared variables

An undeclared value has no var, let or cost modifier and just takes an initial value. Undeclared variables are implicitly scoped to the global scope. They are disallowed in strict mode which is anywhere we use ES6 modules. Frankly it is beyond me why anyone would ever do this ancient crap but I suppose it could come up in interview so here it is.

```
function f()
{
    // Undeclared variable. Implicit global scope
    a = 10;

    // Scope of enclosing function
    var b = 15;
}

f();

console.log(a)

>> 10

JavaScript
```

Undeclared variables do not exist until after they have been assigned to, so the following is a reference exception.

```
console.log(a);
>> Reference error: a is not defined
```

Global

Using ES6 modules and classes massively reduce our exposure to the global scope. Each module imports what it needs directly from other modules. Nevertheless, there are still things we need from the global scope. When running in a browser we use the global scope to access the DOM and the window.

Scope and Looping

There can be some surprising behaviours when we consider loops, especially for loops. Consider the following piece of code.

```
for (const i = 0; i < 3; i++)
     console.log(i);

>> Uncaught TypeError: Assignment to constant variable
```

The problem is JavaScript treats this code like this. The generated variable `ii` is at the outer scope and is hence being redefined.

```
const ii = 0
for (; ii < 3; ii++)
{
    const i = ii;
    console.log(i);
}
```

This is not a problem if we use a for of loop (or indeed a for in loop)

```
for (const i in [0,1,2])
    console.log(i);
```

Functions

Functions are first class objects in JavaScript and there are many ways of creating them.

Function Declarations

The first form we will look at is called a function declaration and it is treated specially. A function declaration is hoisted to the top of its scope. At the same time as it is hoisted it is initialized with its definition. This enables us to execute the following code.

```
console.log(declaredAdd(5,6));

function declaredAdd (a,b)
{
    return a + b;
}

>> 11
```

The declared function itself lives in the scope it is declared it.

```
function declaredAdd(a,b)
{
    return a + b;
}

console.log(declaredAdd);

>> ƒ declaredAdd(a,b)
```

Like var, function declarations belong to the nearest enclosing function scope or the global scope. They never belong to a block scope.

Named Function Expressions

Now let us consider a named function declaration. Although the variable we assign the function expression to (`funcVar`) exists in the containing global scope, the actual function itself (`namedAdd`) only exists inside the function itself.

```
var funcVar = function namedAdd(a,b)
{
    console.log(namedAdd);
    return a + b;
}

funcVar(5,6);

console.log(namedAdd);

>> ƒ namedAdd(a,b)
>> ReferenceError: namedAdd is not defined
```

Anonymous Function Expressions

Anonymous function expressions have no name.

```
var funcVar = function(a,b)
{
    return a + b;
}

console.log( funcVar(5,6));
```

Arrow Functions

ES6 introduces arrow functions. Arrow functions are anonymous and do not use the function keyword at all.

```
var arrowVar = (a,b) => a+b;
```

Arrow functions do not have a prototype and hence cannot be used as constructor functions.

Constructors

If a function invocation is preceded by the new keyword, then the invocation has constructor semantics. Constructor semantics treat arguments, invocation context and return value in a different way from normal function invocations. When invoked as a constructor a new object is created whose prototype is the same as the function's prototype property. Every function has its own prototype, except arrow functions which have no prototype and cannot be used as constructors. The new object is set as the functions execution context and so can be accessed as this inside the function body.

```
function g()
{
    console.log(this.f());
}

g.prototype.f = function() {return "Hello"};

new g();
```

VARIADIC FUNCTIONS

Variable parameter lists are supported via the rest parameter syntax. The … syntax gathers multiple function arguments into a single array.

```
function avg(...args) {
    var sum = 0;
    for (let value of args) {
      sum += value;
    }
    return sum / args.length;
  }
  console.log(avg(2,8));
```

If we want to use the elements of an array as function arguments, we can use the spread syntax. The following uses the spread syntax to send all the elements of the array to the variable parameters of avg.

```
var numbers = [10,20,30]
console.log(avg(...numbers));
```

We can also use the spread operator to apply array elements to normal parameters.

```
function add(x,y) {return x+y;}
console.log(add(...[3,5]));
```

CLOSURE

A scope is a set of variable bindings that are valid at a particular point in time. JavaScript functions are first class objects and can hence be passed around. The means it is highly likely that the scope when a function is invoked will be different from the scope when the function was defined. Despite this a JavaScript function always has access to the variables that were in scope when it was defined. The combination of a function object and the scope that existed when it was defined is known as a closure. Closures are a remarkably powerful feature. We know consider some uses of closures.

Private State

The following factory method returns a function that has access to a private variable called count that it captures via a closure. No one else can manipulate this count except the function returned.

```
function factory()
{
    var count = 0;

    return () => count++;
}
var f1 = factory();
var f2 = factory();

console.log(f1());  // > 0
console.log(f1());  // > 1
console.log(f2());  // > 0
console.log(f2());  // > 1
```

While and Var

In the following code the variable index, defined with var, is scoped to the function. As such ever function closes over the same variable and we see the same integer returned from each.

```
function whileLoop()
{
     var functions = [];
     var count = 0;

     while (count < 3)
     {
         var index = count++;
         functions[index] = () => index;
     }

     return functions;
}

whileLoop().forEach(f=> console.log(f()));

>> 2
>> 2
>> 2
```

While and Let

If we change the previous example to use the let keyword, we see a different result. Let has block scope so each function closes over a different variable.

```
function whileLoop()
{
     var functions = [];
     var count = 0;

     while (count < 3)
     {
         let index = count++;
         functions[index] = () => index;
     }

     return functions;
}

whileLoop().forEach(f=> console.log(f()));

>> 0
>> 1
>> 2
```

Let has block scope so each iteration of the loop closes over a different index variable.

For and Var

This result is unsurprising as we know var is function scoped and so there is only one variable captured by all functions.

```
  function loop()
  {
      var functions = [];

      for (var i =0; i<3; i++)
          functions[i] = () => i;

      return functions;
  }

  loop().forEach(f=> console.log(f()));

>> 3
>> 3
>> 3
```

For and Let

Where is the variable `i` scoped? Is it scoped outside or inside the loop body? It turns out it is scoped as being inside the loop body so we get the correct result.

```
  function loop()
  {
      var functions = [];

      for (let i =0; i<3; i++)
          functions[i] = () => i;

      return functions;
  }

  loop().forEach(f=> console.log(f()));
```

```
>> 0
>> 1
>> 2
```

For/Of and Let

Once again, the variable index is scoped inside the loop body, so we get the expected result.

```
function loop()
{
     var indices = [0,1,2];
     var functions = [];

     for (let index of indices)
         functions.push(() => index);

     return functions;
}

loop().forEach(f=> console.log(f()));
```

```
>> 0
>> 1
>> 2
```

For/Of and var

The result is unsurprising.

```
function loop()
{
     var indices = [0,1,2];
     var functions = [];

     for (let index of indices)
         functions.push(() => index);

     return functions;
}

loop().forEach(f=> console.log(f()));

>> 2
>> 2
>> 2
```

Call and Apply

Both methods allow one to indirectly invoke a function. Both functions take as the first argument the object on which they are invoked. The first argument then forms the execution context. Subsequent arguments are then passed as arguments to the function being invoked. The main difference between call and apply is that apply expects the arguments to be an array. The following example will make this clear.

```
let o = {
    x:5
}

function f(a,b)
{
    console.log(`this.x=${this.x} a=${a}, b=${b}`);
}

f.call(o, 10,15);
f.apply(o, [10,15]);
```

If call and apply are used with arrow functions the first argument is ignored as they functions always take the execution context at the point of definition.

Bind

Invoking bind on a function and passing an object argument o creates a new function that has o as its execution context. Bind does not work on arrow functions, but this is generally not a problem as bind is used to make normal functions behave like arrow functions.

```
let o = {
    x:5
}

function f(a,b)
{
    console.log(`this.x=${this.x} a=${a}, b=${b}`);
}

let g = f.bind(o);

g(10,15);
```

Bind can also be used to perform currying or partial application.

```
function add(a,b)
{
    return a + b;
}

let next = add.bind(null,1);
console.log(next(6));
```

Comparisons and coercions

Consider difference between equality and equivalence.

=== disallows type coercison

All other comparison operators perform coercion. This includes <, >, <=, >=, ==

These operators typeically prefer to coerce to numbers where possible

For objects === uses identity comparison rather than structural value comparison

Arrays are objects so also using referential compasrions.

JS has no means of performing structural object comparisons. We need to implement it ourselves.

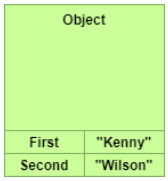
Coercion is when a value of one type is converted to its equivalent representation in another type.

== allows type coercion before the comparions while === does now.

Objects and Prototypes

PROPERTIES

A JavaScript object is essentially a collection of properties where each property associates a key with a value. The key must be a string or a symbol.



Initializing Objects

The following defines an object using literal format.

```
var name = {
    First : "Kenny",
    Second: "Wilson"
}
```

Although the literal format is the preferred way on initializing a new object, we can also use the following form.

```
var o1 = new Object();
o1.First = "Kenny";
o1.Second = "Wilson";
```

We can also use variables to initialise object properties. The property name takes the variable's name, and the property value takes the variable's value.

```
var first = "Kenny";
var o2 = {first};
console.log(o2);

>> {first:"Kenny"}
```

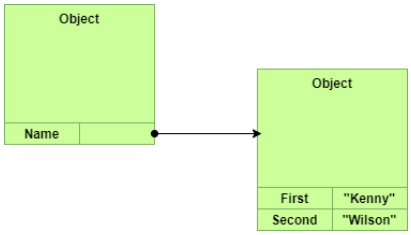
Accessing Object Properties

Given an object we can access its properties use the '.' operator or by using a string key.

```
// Accessing using '.' operator
console.log(o3.Name.FirsName);

// Accessing using string keys
var key = "Name";
console.log(o3[key]["SecondName"]);
```

Complex Properties

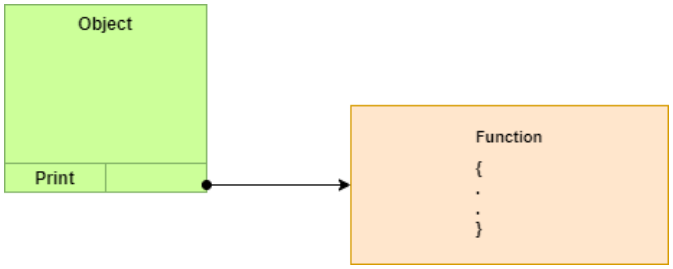
An object's property values can themselves be other objects giving us complex properties.




```
var o3 = {
    Name :
    {
        First: "Kenny",
        Second: "Wilson",
    },

}
```

METHODS

Because functions are first class objects, we can set an object's property value to be a function thereby creating a method.

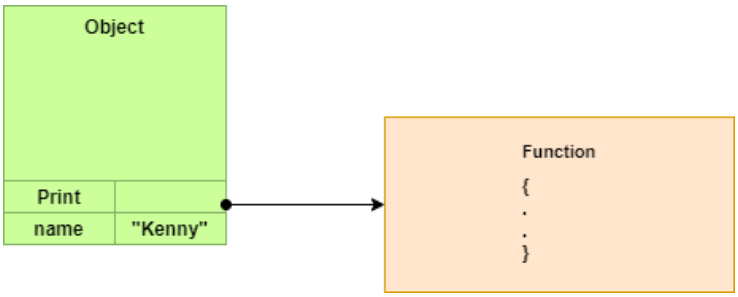



```
var o5 = {
     Print: function () {
         console.log("Hello World");
     }
}

o5.Print();
```

Execution Context (this)

In JavaScript, the this keyword refers to an execution context. It allows us to carry out the following.

```
var o5 = {
    name:"Kenny",
    print: function() {
        console.log("Hello World " + this.name);
    }
}

o5.print();

>> Hello World Kenny
```



We need to be careful. When invoked through the `o5` object using the '.' operator, `this` is defined to be the object `o5`. If we store the method in a variable as follows `this` is undefined (in non-strict contexts it bizarrely take the global object).

```
var o5 = {
    name:"Kenny",
    print: function() {
        console.log("Hello World " + this.name);
    }
}

var f = o5.print;
f();

>> Hello World undefined
```

Inner Function and Execution Context.

Inner functions do not automatically take the execution context of their enclosing method.

```
var o5 = {
     name:"Kenny",

     method: function() {

          return function()
          {
               console.log("Hello World " + this.name);
          }
     }
}

o5.method()();
```

Arrow functions however behave differently and take the execution context at the point they are defined. This is in fact one of the defining characteristics of arrow functions.

```
var o6 = {
     name:"Kenny",

     method: function() {

          return () =>console.log("Hello World " + this.name);
     }
}

o6.method()();
```

Bind

To fix the problem mentioned in the previous section we can bind an execution context to a method.

```
var o5 = {
     name:"Kenny",
     print: function() {
          console.log("Hello World " + this.name);
     }
}

var f = o5.print.bind(o5);
f();

>> Hello World Kenny
```

Mind Bending Bind

A little mind bending is the following. Because the function f is taking the global execution context and because undeclared variables belong to the global context, we get the following.

```
var o5 = {
    name:"Kenny",
    print: function() {
        console.log("Hello World " + this.name);
    }
}

name= "John";
var f = o5.print;
f();


>> Hello World John
```

Call

An alternative to bind is to use the call method on the function to pass in an execution context.

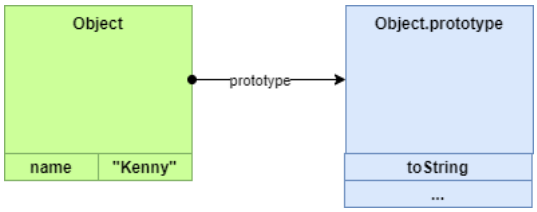
```
var f = o5.print.bind(o5);
f.call(o5);
```

PROTOTYPES

Unless explicitly disabled, all objects in JavaScript have a prototype which provides a set of common methods. Consider the following piece of code.

```
var o5 = {
    name:"Kenny",
}
```

JavaScript will implicitly set the prototype of this object to be the default prototype.



This prototype, which can be accessed explicitly as `Object.prototype`, provides common behaviour such as the `toString` method. When we access a property on our object, JavaScript will first look for a definition on the object itself and if none is found it then looks on the prototype. Invoking toString on our object o5 uses the method defined on `Object.prototype`

```
console.log(o5.toString());
```

```
>> [object Object]
```

We can access the prototype of an object using the `Object.getPrototype` method. The following statement shows our object o5 has a prototype which is the default object prototype `Object.prototype`.

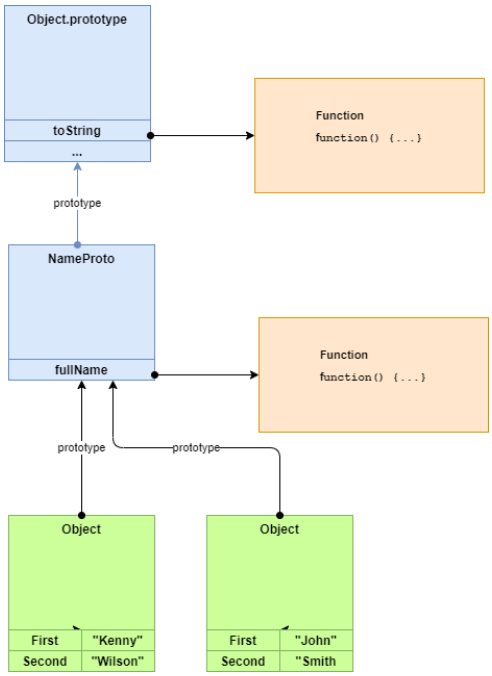
```
console.log(Object.prototype == Object.getPrototypeOf(o5));
```

```
>> true
```

Custom Prototypes

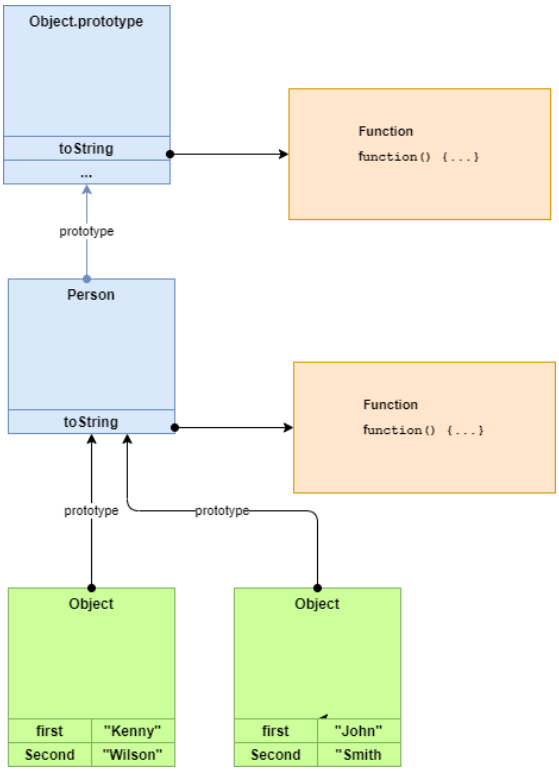
We can create our own prototypes to provide common behaviour across a set of objects. The following code fragment shows how to create such a prototype. Note the importance of the execution context `this`. Each object provides its own execution context `this` We can see from this code that a prototype is just a standard JavaScript object.

```
var a = {
    first:"Kenny",
    second:"Wilson",
}
var b = {
    first:"John",
    second:"Smith",
}

var NameProto = {
    fullName: function() {
        return this.first + " " + this.second;
    }
}

Object.setPrototypeOf(a,NameProto);
Object.setPrototypeOf(b,NameProto);

console.log(a.fullName());
console.log(b.fullName());

>> Kenny Wilson
>> John Smith
```



Overriding Methods

We saw that `Object.prototype` provides a `toString` method. If we want our own implementation, we can add a method to our custom prototype. Because the method resolution walks back up the prototype chain it will find our method first and hence that will override the one further up the chain in Object.prototype.

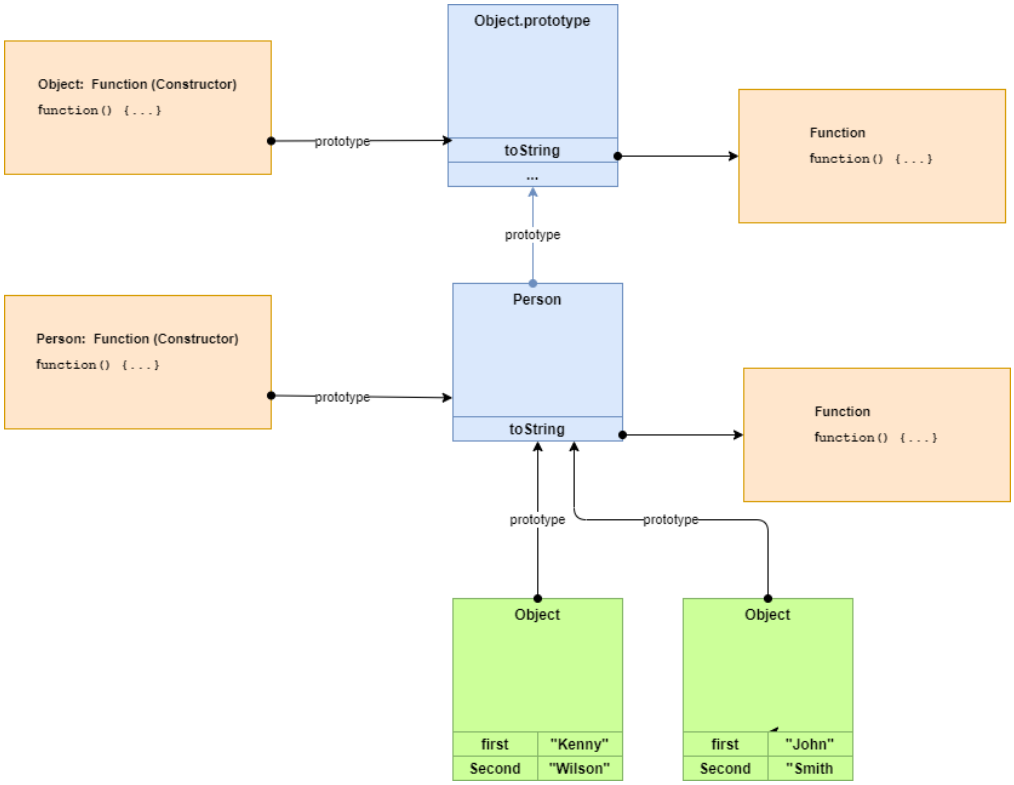



```
var a = {
    first:"Kenny",
    second:"Wilson",
}
var b = {
    first:"John",
    second:"Smith",
}

var Person = {
    toString: function() { return this.first + " " + this.second }
}

Object.setPrototypeOf(a, Person );
Object.setPrototypeOf(b, Person );

console.log(a.toString());

>> Kenny Wilson
```

CONSTRUCTORS

The code in the previous section was a little longwinded. Each object must duplicate the field initialization logic and we need to explicitly set the prototype on each object. The answer is to use a function as a constructor. Every function, except arrow functions, have a prototype property. When a function is invoked using the new keyword it creates a new object and sets that new object's prototype to be the same prototype as the function's own prototype property. This created object is assigned to the execution context and hence this can be used in the function body to initialize the new object.

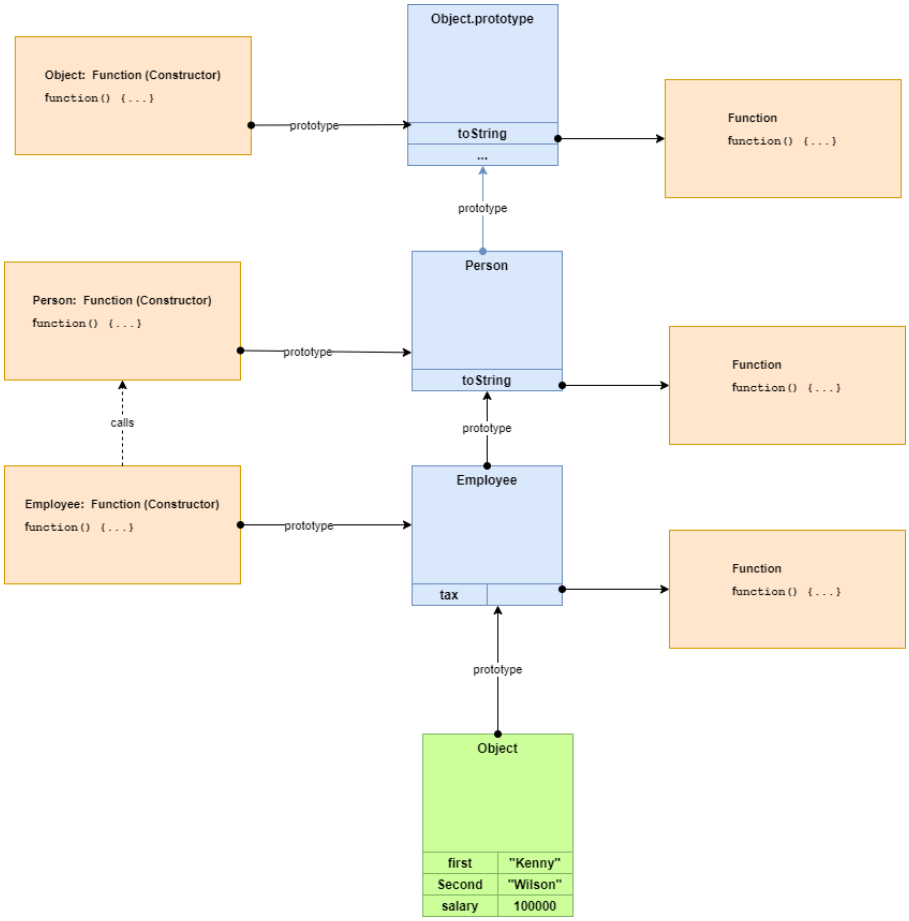
```
let Person = function(first, second)
{
    this.first = first;
    this.second = second;
}

let a = new Person("Kenny", "Wilson");
let b = new Person("John", "Smith");

console.log(a.toString());
```



Constructor Chaining

We need to be careful when chaining constructors.

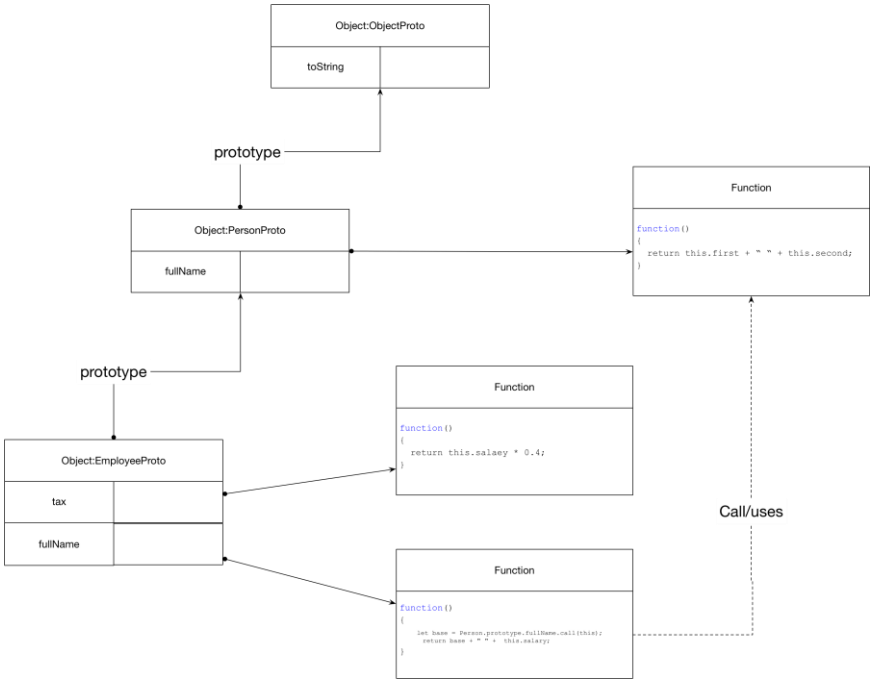
```
let Person = function(first,second)
{
    this.firstName = first;
    this.secondName = second;
}

Person.prototype.fullName = function()
{
    return this.firstName + " " + this.secondName;
}

let Employee = function(first,second,salary)
{
    Person.call(this,first,second);
    this.salary = salary;
}

Object.setPrototypeOf(Employee.prototype,Person.prototype);
Employee.prototype.tax = function () {return this.salary * 0.4};

 var emp = new Employee('Kenny', 'Wilson',100000);
 console.log(emp.fullName());
 console.log(emp.tax());

>> Kenny Wilson
>> 40000
```



Overridden methods

When we override a method, we might want the overridden method to be called from the new method.

```
Employee.prototype.fullName = function() {
    let base = Person.prototype.fullName.call(this);
    return base + " " +  this.salary;
}

 var emp = new Employee('Kenny', 'Wilson',100000);
 console.log(emp.fullName());
 console.log(emp.tax());

>> Kenny Wilson 100000
>> 40000
```



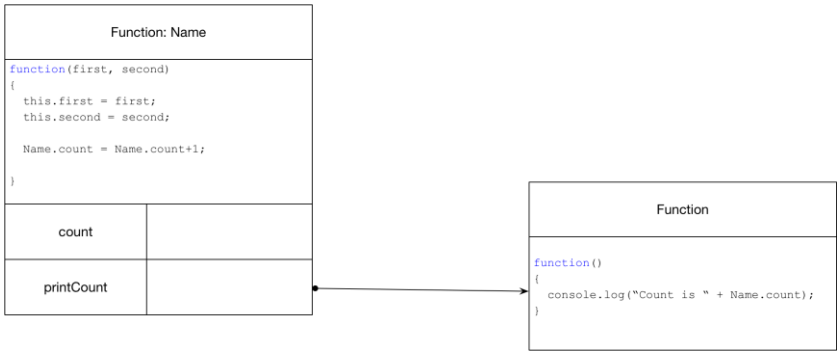
Instanceof

```
var emp = new Employee('Kenny', 'Wilson',100000);
console.log(emp instanceof Person);
console.log(emp instanceof Object);

>> true
>> true
```

Static Properties and methods

Static properties and methods are just properties and methods on the Constructor function object.

```
let Name = function(first, second)
{
    this.first = first;
    this.second = second;

    Name.count = Name.count+1;
}

Name.prototype.fullName = function() {
    return this.first + " " + this.second;
}

// Add a static variable to the Name type
Name.count = 0;

// Add a static method
Name.printCount = function() {
    console.log("Count is " +Name.count);
}

console.log(Name);

>> [Function: Name] { count: 2, printCount: [Function] }
```



Getters and Setters

We can create getters and setters as follows.

```
Object.defineProperty(Car.prototype,"length", {
    get: function() {return this._length;},
    set: function(length) {this._length=length;}
});
car.length = 100;
console.log(car.length);
```

PROPERTIES

Checking Existence

The `in` operator checks if a property exists in an object or in any of the prototypes in its prototype chain.

```
let aname = {
    first : "Kenny"
}

console.log("first" in aname);
console.log("toString" in aname);

>> true
>> true
```

The method `hasOwnProperty` checks if a property exists in an object. The prototype chain is not checked.

```
console.log(aname.hasOwnProperty("first"));
console.log(aname.hasOwnProperty("toString"));

>> true
>> false
```

Property Attributes

Properties specified in an object literal are writable and enumerable. If we want our properties to have different attributes, we can use the `Object.define` function. The full list of attributes is listed here. The following shows how to create a read-only, non-enumerable property.

```
let n = {
    first: 'kenny',
    second: 'wilson'
}

Object.defineProperty(n, 'age', {
    value: 42,
    writable: false,
    enumerable: false
  });
```

Enumerating properties

Consider the following object.

```
let n = {
    first: 'kenny',
    second: 'wilson'
}

Object.defineProperty(n, 'age', {
    value: 42,
    writable: false,
    enumerable: false
});
```

We can list the enumerable properties as following.

```
console.log(...Object.keys(n)); // first second
```

We can include the enumerable properties as following.

```
console.log(...Object.getOwnPropertyNames(n)); // first second age
```

Arrays

Arrays in JavaScript can be spare. Consider the following examples.

```
let a3 = [];
a3[2] = 'a';
console.log(...a3);      // undefined undefined a
console.log(a3.length); // 3

let a4 = ['a'];
a4.length = 3;
console.log(...a4);      // a undefined undefined

let a5 = ['a','b','c'];
a5.length = 1;
console.log(...a5);      // a
```

ADDING TO FRONT/BACK

We use `push` to add a new element at the end of the array and `unshift` to add a new element at the beginning of the array.

```
let a = ['c'];
a.push('d')
console.log(...a); // c d

a.unshift('b');
console.log(...a); // b c d
```

Modules

ES5 Vs Es6

JavaScript modules have changed a lot down the years. Let us look at modules in es5 and es6.

ES5

The code is listed here. Run the example by typing the following command from the directory that contains `modconsumer.js`

```
node modconsumer.js
```

Listing 1 es5mod.js

```
var myModule = {
    add: function(x,y) { return x+y},
    sub: function(x,y) { return x-y}
}

module.exports = myModule;
```

Listing 2 modconsumer.js

```
var myModule = require('./es5mod');

console.log(myModule.add(10,20));
```

ES6

ES6 modules, also known as ESM, are file based. One file contains one module. We do not have to explicitly use strict mode with ES6 modules because they are automatically strict mode. Unlike older module formats ES6 modules do not require instantiation. Instead, we just import its exported symbols. ES6 modules are effectively singletons in that there is only ever one instance created.

If we only use ES6 modules each module imports references from other ES6 modules it needs, thereby minimising any interaction with the global scope which is a huge advantage.

The code is listed here. Run the example by typing the following command from the directory that contains `modconsumer.js` Note that to run ES6 modules using node we need to add `"type": "module"` to the `package.json` at the top level.

```
node modconsumer.js
```

Listing 3 es6mod.js

```
var myModule = {
    add: function(x,y) { return x+y},
    sub: function(x,y) { return x-y}
}

export default myModule;
```

Listing 4 modconsumer.js

```
import myModule from './es6mod.js';

console.log(myModule.add(10,20));
```

SINGLETON

If multiple modules include the same module then only one copy is created so we can export consts from a module to create effective singletons.

See this example

Types

JavaScript has a simple Type System consisting of the following types.

- string
- number
- boolean
- undefined
- null
- object

Undefined is the default value for uninitialized variables. Undefined is also a data type. Null is also a data type.

Truth

Name	Code
0	`false`
Non-zero number	`True`
Undefined	`false`
NaN	`false`
{}	`true`
[]	`true`
Null	`false`
Null	`False`

Numbers

64 bit IEEE 754 standard floating point numbers which support integers from -2^53 to 2^53 inclusive.

Dates

Number of milliseconds since January 1st 1970.

String

Mozilla String Reference

Immutable sequence of Unicode values. There are no characters in JavaScript only strings with one element.

Operators

!! Convert to bool.

```
> !!''
> false
```

|| to provide default value.

```
>> const v = input || "kenny"
>> undefined
>> v
>> "kenny
```

Equality

JavaScript provides the double equals operator == and the triple equals operator === . The double equals operator uses type coercion where the two operands have different types. The triple equals operator does not use type correction. Where both arguments are of the same type both operators behave the same. Primitive types use value comparison.

```
let c = "Kenny";
let d = "Kenny";
console.log(c == d);
console.log(c === d);

>> true
>> true
```

Object types use reference comparison.

```
let a = { name:"Kenny"};
let b = { name: "Kenny"};
console.log(a == b);
console.log(a === b);

>> false
>> false
```

Arrays are objects

```
let a = [1,2,3];
let b = [1,2,3];
console.log(a == b);
console.log(a === b);

>> true
>> true
```

The double equals performance type correction.

```
console.log( "1" == 1);
console.log( "1" === 1);

>> true
>> false
```

Iterators, Iterables and Generators

The built-in types string, array, Map, Set and all iterable which means they can be consumed by for/of loops and rest operators.

```
let a = [4,8,12];

for (let idx of a)
{
    console.log(idx);
}

console.log(...a)
```

In this section we look at how iterables work. There are four key concepts when working with iterables.

- Iterable
- Iterator
- Iterator Result
- Generator

Iterator and Iterator Result

An iterator is an object that contains a next method that returns iterator result object. An iterator is ab object with two properties: `done` and `value`. Done indicates if the sequence is finished and value gives the value if the sequence is not complete.

```
{done:false,value:this.count}
```

Consider the following example.

```
let iterator = {

    count:0,

    next: function(){

        let res = (this.count < 2) ?
        {done:false,value:this.count} :
        {done: true,value:undefined }

        this.count = this.count+1;
        return res;
    }
}

 console.log(iterator.next());
 console.log(iterator.next());
 console.log(iterator.next());
```

Running the example we see the following output.

```
>> { done: false, value: 0 }
>> { done: false, value: 1 }
>> { done: true, value: undefined }
```

ITERABLE

An iterable object is an object with a special method that returns an iterator. The following shows the relationship between an iterator and an iterable. Note the method that produces the iterator has the special key `[Symbol.iterator]`.

```
let iterator = {

    count:0,

    next: function(){

        let res = (this.count < 2) ?
        {done:false,value:this.count} :
        {done: true,value:undefined }

        this.count = this.count+1;
        return res;
    }
}

 let iterable = {
    [Symbol.iterator]: function() { return iterator; }
 }

console.log(...iterable);

>> 0 1
```

PUTTING IT TOGETHER

An iterable can be consumed by the spread operator. The following shows all three parts: iterator, iterable and consumer.

```
// Iterator
let Iterator = function(count) {
    this.count = count;

    this.next = function() {
        let res = (this.count >=0) ?
        {done:false,value:this.count} :
        {done: true,value:undefined }

        this.count = this.count-1;
        return res;
    }
}

// Iterable
let Iterable = function(count) {
    this.count = count;

    this[Symbol.iterator] = function() {
        return new Iterator(this.count);
    }
}

// Consumer of Iterable
console.log(... new Iterable(2));
```

GENERATORS

The Language has support for generating Iterable. We can simplify our Iterable from the previous example and get rid of the explicit Iterator object.

```
let Iterable = function(count) {
    this.count = count;

    this[Symbol.iterator] = function() {
        return new Iterator(this.count);
    }
}

// Consumer of Iterable
console.log(... new Iterable(2));
```

Iterable Types

The following language types are all iterable.

- String
- Array
- Map
- Set

Most build in iterables support iterating key, values, or entries.

```
let a = [4,8,12];

for (let idx of a.keys())
{
    console.log(idx);
}

for (let value of a.values())
{
    console.log(value);
}

for (let [idx,value] of a.entries())
{
    console.log(idx, "=",value);
}
```

CONSUMING ITERABLES

For of

```
for(let a of [1,2,3])
    console.log(a*2);

>> 2
>> 4
>> 6
```

Spread into function with rest arguments.

```
console.log(...[1,2,3]);

>> 1 2 3
```

Spread into function with normal arguments.

```
let a = [1,2,3];

add = (a,b,c) => a+b+c;
console.log(add(...a));
```

Destructuring Assignment

```
let [a, b, c] = new Set(['a', 'b', 'c']);
console.log(a);

>> a
```

Yield*

```
let f = function*()
{
    return yield* [1,2,3]
}

console.log(...f());

>> 1 2 3
```

For in

The for in construct iterates all enumerable properties. When we add properties to an object by just assigning to them, they are by default enumerable.

```
var a = {
    first : "k",
    second: "w"
}

for (var name in a) {
    console.log(name);
}

>> first
>> second
```

If we want non-enumerable properties, we can use the `Object.defineProperty` method.

```
var a = {
    first : "k",
    second: "w"
}

Object.defineProperty(Object.prototype,
    "notshown",
    {enumerable:false,value:"good"});

for (var name in a) {
    console.log(name);
}


>> first
>> second
```

If we want to test whether a property is on the object itself and not coming from one of its prototypes, we can use `hasOwnProperty`

```
var a = {
    first : "k",
    second: "w"
}

console.log(a.hasOwnProperty("first"));
console.log(a.hasOwnProperty("toString"));

>> true
>> false
```

Promises

Creating Promises

In order to understand how Promises work, the best way is to write a function that creates one from scratch. The function below simulates a long running operation that returns a message.

```
function futureMessage(time, message)
{
    ❸function promiseControl(resolveHandle, rejectHandle)
    {

        setTimeout(() => {
            ❺resolveHandle(message)
        }, time);
    };

    return new Promise(❷promiseControl);
}

Let ❶ promise = futureMessage(1000, "Message One");

let promise2 =  promise
    ❹.then(❻m => console.log(`received ${m}`));
```

Let us now go over the code fragment and describe what happens.

❶ **Invoke function**	We invoke our long running operation and pass a piece of state to represent a message.
❷ **Create Promise**	We pass the control function to the Promise
❸ **Promise calls back**	The promise immediately and synchronously calls our function. Now our function can start its operation. The promise passes us two handles that we use to let the promise know the operation has completed successfully or caused an error.
❹	We register a callback to be invoked when the promise completes successfully.
❺	We resolve the promise
❻	We handle the callback from the promise.

Chained Promises

We can chain three promises in a clunky form to show how it works.

```
let promiseOne = futureMessage(2000, "One");

let promiseTwo = promiseOne.then(m => {
    console.log(`Received message ${m}`);
    return futureMessage(2000, "Two");
});

let promiseThree = promiseTwo.then(m => {
    console.log(`Received message ${m}`);
    return futureMessage(2000, "three");
});

promiseThree.then(m=> console.log(`Received message ${m}`));

>> Received message One
>> Received message Two
>> Received message Three
```

Tidying up the syntax

Notice we don't even store any of the promises.

```
futureMessage(2000, "One")
.then((m => {
    console.log(`Received message ${m}`);
    return futureMessage(2000, "Two");
}))
.then((m => {
    console.log(`Received message ${m}`);
    return futureMessage(2000, "Two");
}))
.then(m=> console.log(`Received message ${m}`));
```

Error Handling

Let us change our code to provide an error and handle it. Note that we add the catch at the end and it handle any errors anywhere in the chain.

```
futureMessage(2000, "One")
    .then((m => {
        console.log(`Received message ${m}`);
        return futureMessage(2000, "Two");
    }))
    .then((m => {
        throw new Error("Details of error")
    }))
    .then(m => console.log(`Received message ${m}`))
    .catch(e => {
        console.log(`Caught error ${e}`);
    });
```

ASYNC AWAIT.

We can tidy the code up using async/await. Let us adjust the code to throw errors if a flag is included.

```
function futureMessage(time, message, isError) {
    // When creating a Promise from scratch we need
    // to declare a function that will control the
    // Promise. We pass this function to the Promise's
    // constructor and the constructor invokes it
    // synchronously in order to pass us two objects
    // we can use to signal a normal or error completion.
    //
    // Note by defining this function inside the outer function
    // we are using a closure to pass more into the function.
    function promiseControl(resolveHandle, rejectHandle) {
        setTimeout(() => {
            isError ?  rejectHandle(message) : resolveHandle(message)
        }, time);
    };

    return new Promise(promiseControl);
}
```

Now let's deal with an error.

```
async function f()
{
    try
    {
        let m = await futureMessage(2000, "One");
        console.log(`Received message ${m}`);
        m = await futureMessage(2000, "Two");
        console.log(`Received message ${m}`);
        m = await futureMessage(2000, "Three", true);
        console.log(`Received message ${m}`);
    }
    catch (e)
    {
        console.log(`error hapnd ${e}`)
    }
}

f();

>> Received message One
>> Received message Two
>> error hapnd Three
```

Exceptions

```
throw {message: 'a'};
```

Questions – The Type System

Collections

Object as Symbol Table (Keys must be Strings)

```
var st = {
    kenny: new Employee('Kenny', 'Wilson', 100000),
    sanna:  new Employee('Sanna', 'Hulkki', 40000)
};

console.log(Object.keys(st));
console.log(Object.values(st));
```

Collections

Any object in JavaScript can be used as a symbol table. The keys in an object are always strings.

```
var st = {
    kenny: new Employee('Kenny', 'Wilson', 100000),
    sanna:  new Employee('Sanna', 'Hulkki', 40000)
};

console.log(Object.keys(st));
console.log(Object.values(st));

>> [ 'kenny', 'sanna' ]
>>[
>>  Employee { firstName: 'Kenny', secondName: 'Wilson', salary: 100000 },
>>  Employee { firstName: 'Sanna', secondName: 'Hulkki', salary: 40000 }
>>]
```

We can also use a Map as a symbol table when the keys are not strings. It also has a Set which prevents duplicates.

Lists/Arrays

Modules

JavaScript modules have changed a lot down the years

ES5

Listing 5 es5mod.js

```
var myModule = {
    add: function(x,y) { return x+y},
    sub: function(x,y) { return x-y}
}

module.exports = myModule;
```

Listing 6 modconsumer.js

```
var myModule = require('./es5mod');

console.log(myModule.add(10,20));
```

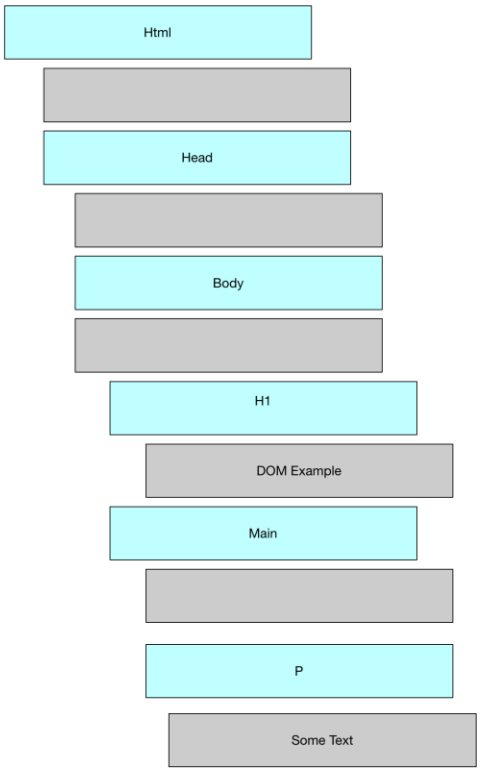
ES6

```
var myModule = {
    add: function(x,y) { return x+y},
    sub: function(x,y) { return x-y}
}

export default myModule;
```

The Browser

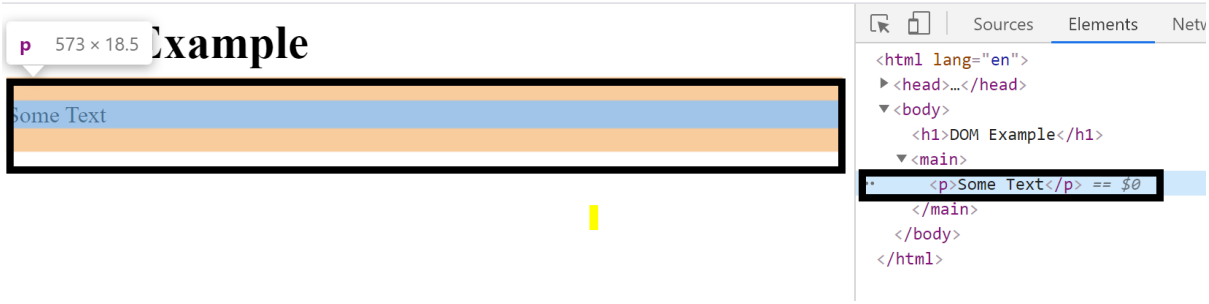
The DOM

```
<html lang="en">
<head>
    <title>Document</
title>
</head>
<body>
    <h1>DOM Example</h1>
    <main>
        <p>Some Text</p>
    </main>
</body>
</html>
```



Chrome Dev Tools

Go to the Elements tab in the dev tools and you will see the DOM tree. Select any node to see it highlighted on the rendered web page.



Notice the selected node has $0 beside it. We can use this to reference this node from the Console.

```
> $0
> <p> Some Text <p>
```

Notice that when we select the document, we get a view of the html in the console. And again, we can select sub-nodes in this document to see the actual rendered object on the web page.

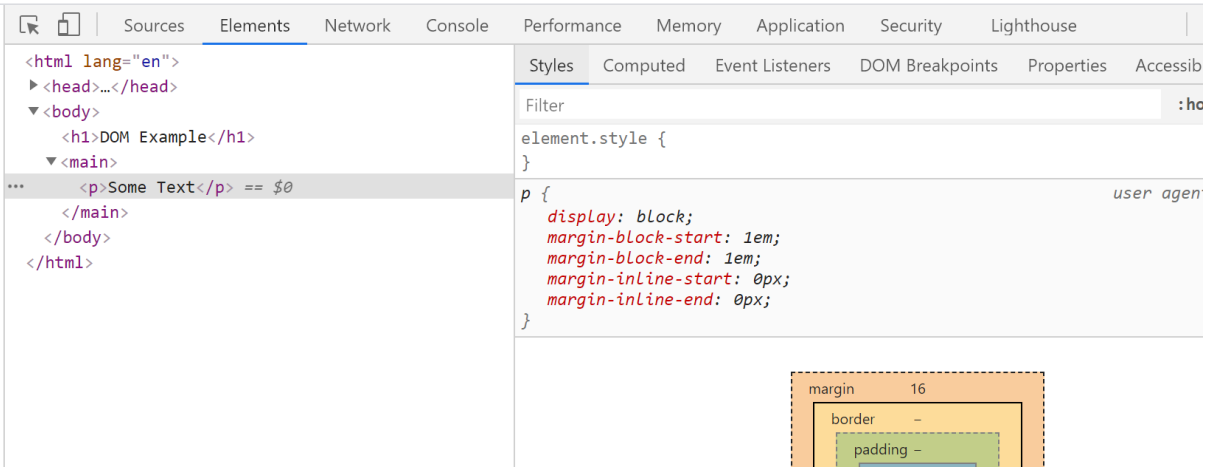
```
> document
  ▼#document
      <html lang="en">
      ▼<head>
          <title>Document</title>
        </head>
      ▼<body>
          <h1>DOM Example</h1>
        ▼<main>
            <p>Some Text</p>
          </main>
        </body>
      </html>
>
```

If we want to see the node as an object with its properties use the dir command.

```
> dir(document)
```

```
▼#document
   URL: "file:///C:/Users/rps/Code/temp/domexample/index.html"
 ▶ activeElement: body
 ▶ adoptedStyleSheets: []
   alinkColor: ""
 ▶ all: HTMLAllCollection(7) [html, head, title, body, h1, main, p]
 ▶ anchors: HTMLCollection []
 ▶ applets: HTMLCollection []
   baseURI: "file:///C:/Users/rps/Code/temp/domexample/index.html"
   bgColor: ""
 ▶ body: body
   characterSet: "windows-1252"
   charset: "windows-1252"
   childElementCount: 1
 ▶ childNodes: NodeList [html]
```

When we have selected a node in the Element tab, we can see the styles and event listeners to the side.



Questions – Language Basics

Type

What are the types supported by JavaScript?

Boolean

String

Number

Undefined

Object

Null

What are the special type of objects?

Arrays and object

VARIABLES AND SCOPE

`…\Code\bitbucket\webdev\exposition\javascript\Interview Questions\2. Scope`

What keywords can we use to scope variables?

let,var, const

What is the effect of var?

Creates a lexically scoped variable

What does this mean?

The variable is scoped by its execution context

Blocks have no impact on scope

If a variable is declared inside a function with var what is its execution context?

The enclosing function

If a variable is declared outside a function with var what is its execution context?

The global context

When a variable is declared with var outside of all functions the execution context is the global context

What is an undeclared variable?

A variable that is not declared with any scope modifier and is just initialised with a value

What is the scope of an undeclared variable?

The global context.

What is output of this code and why?

```
function f()
{
    // Undeclared variable. Implicit global execution context
    a = 10;

    // Execution context of enclosing function
    var b = 15;
}

f();

console.log(a)
```

10 is output because the variable a is undeclared and hence takes the global context

What is output of this code and why?

```
a = 5;

delete a;

console.log(a);
```

5

ReferenceError: a is not defined

The reason is undeclared variables can be deleted.

What is output of this code and why?

```
a = 10;
console.log(10);

var a;

>> 10
```

Declared variables are declared as if the statement was at the top of the file

What is this known as?

hoisting

What is output of this code and why?

```
{
    var a = 5;
}

console.log(a);
```

5. Because blocks have no impact on the scope of var declared variables.

What is output of this code and why?

```
{
    var a = 5;
}

var a;

console.log(a);
```

5 because re-declaring has no effect and does not clear the variable

How are variables with block scope declared?

Using let

What is output of this code and why?

```
{
    let mylex = 4;
}

console.log(mylex);
```

ReferenceError: mylex is not defined.

What is output of this code and why?

```
var x = 5;
var y = 6;

 {
  var x = 10;
  let y = 11;

  console.log(x);
  console.log(y);
}

console.log(x);
console.log(y);

>> 10
>> 11
>> 10
>> 6
```

What is output of this code and why?

```
let x = 1;

{
  var x = 2;
}

>> SyntaxError
```

CLOSURES

EQUALITY

What is the result of this code?

```
let s = "10";
let t = 10;
console.log(s==t);
```

true

Why?

Type coercion

FUNCTIONS

What is the difference between these two forms?

```
var sum = function(a, b)  {return a + b;}

function add(a,b) { return a+b;}
```

The first form is an anonymous function expression. The second form is a function declaration. Function declarations can be used before the point in the file where they are declared because they are hoisted.

How do arrow functions differ from other function forms?

They do not have a prototype and so can't be used as constructors.

The `this` keyword takes the execution context at the point they are defined.

Consider the following code. What is output and why?

```
let object =  {

    method: function()
    {
        let mythis = this;
        console.log(this === object);

        inner();

        function inner()
        {
            console.log(object === this);

        }
    }
}
```

>> true

>> false

Inner functions do not inherit the execution context of their containing function.

What does the length property of a function show?

The arity or number of parameters. Rest parameters do not count towards the length

Consider the following code. What is output and why?

```
let object =  {

    method: function()
    {
        let mythis = this;
        console.log(this === object);

        const inner = () => {
            console.log(object === this);
        }

        inner();
    }
}
```

>> true

>> true

Because unlike normal nested functions, arrow functions take the execution context at the point they are defined.

Which functions do not have a prototype?

Arrow functions

OBJECTS

What is the output of the following and why?

```
var calculator = {
    a: 10,
    b: 20,
    sum() {
        return this.a + this.b;
    },
};

var f = calculator.sum;
console.log(f());
```

NaN because this is not bound at the point that f is invoked because it is not invoked through the object.

Fix the code so it works

```
var calculator = {
    a: 10,
    b: 20,
    sum() {
        return this.a + this.b;
    },
};

var f = calculator.sum.bind(calculator);
console.log(f());
```

What is the output of the following and why?

```
var calculator = {
    a: 10,
    b: 20,
    sum: () => this.a + this.b
};

console.log(calculator.sum());
```

NaN because the method is defined as a property which returns a lambda. The lambda has no outer function in which this is defined. So this is not defined when it is invoked.

Questions – Miscellaneous

What is transpiling?

Using at tool to covert source code to another textual source code form.

How can forward compatibility be achieved?

Transpiling newer language syntax to older language forms.

Using shims for missing API's

Do I need to turn on strict mode inside ES6 modules?

No. All ES6 modules automatically assume strict mode.

Development Environment

Specified Single File

Run

Open a terminal and enter the command.

```
node hello.js
```

Run and Watch

Setup package.json if you have not already

```
npm init --yes
```

Install the `nodemon` node package as a development dependency.

```
npm install --save-dev nodemon
```

If we want to run the dev dependency from the terminal we use the `npx` command

```
npx nodemon hello.js
```

Run as Script

As we install it as a dev dependency, we can only run it from the scripts section of package.json

```
{
  "name": "JS",
  "version": "1.0.0",
  "description": "",
  "main": "test.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "watch" : "nodemon hello.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "install": "^0.13.0",
    "nodemon": "^2.0.4",
    "npm": "^6.14.8"
  }
}
```

Run the script `npm run watch`

Debug

```
{
      "version": "0.2.0",
      "configurations": [
          {
              "type": "node",
              "request": "launch",
              "name": "Launch Program",
              "skipFiles": [
                  "<node_internals>/**"
              ],
              "program": "${workspaceFolder}\\hello.js"        }
      ]
  }
```

You can now run or debug the file which has focus by using the command `Ctrl-F5` or `F5` respectively on windows.

Debug with Watch

Setup a launch.json target as follows. Make sure nodemon is installed globally

```
          {
              "name": "Launch server.js via nodemon",
              "type": "node",
              "request": "launch",
              "runtimeExecutable": "nodemon",
```

```
            "program": "${workspaceFolder}/hello.js",
            "restart": true,
            "console": "integrated
Terminal",
            "internalConsoleOptions": "neverOpen"
        }
```

Now run or debug it using Ctrl-F5 or F5 respectively

For more details see

https://code.visualstudio.com/docs/nodejs/nodejs-debugging

Currently Selected File

DEBUG

Add the following to your launch.json

```
{
      "version": "0.2.0",
      "configurations": [
          {
              "type": "node",
              "request": "launch",
              "name": "Launch Program",
              "skipFiles": [
                  "<node_internals>/**"
              ],
              "program": "${file}"        }
      ]
  }
}
```

Now use Ctrl-F5 or F5 to run or debug the currently selected file

Tests

Run All Tests

First, we install jest

```
  npm install --save-dev jest
```

Now we can run all the tests as

```
npx jest
```

Run single test file

```
npx jest myModule.test
```

Run Specified Test

```
npx jest myModule.test -t=<TestName>
```

Run all tests in Debug Mode

Add the following to vs code on Mac and run debug from the VS Code console. You will need something else on windows.

```
{
    "name": "Debug tests single run",
    "type": "node",
    "request": "launch",
    "env": { "CI": "true" },
    "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/jest",
    "args": ["test", "--runInBand", "--no-cache"],
    "cwd": "${workspaceRoot}",
    "protocol": "inspector",
    "console": "integratedTerminal",
    "internalConsoleOptions": "neverOpen"
}
```

Run Single Test File in Debug Mode

```
{
    "name": "Debug single tests single run",
    "type": "node",
    "request": "launch",
    "env": { "CI": "true" },
    "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/jest",
    "args": ["--runInBand", "--no-cache"],
    "cwd": "${workspaceRoot}",
    "program": "${fileBasenameNoExtension}",
    "protocol": "inspector",
    "console": "integratedTerminal",
    "internalConsoleOptions": "neverOpen"
}
```

Run Single Test File in Debug Mode With Watch

```
{
    "name": "Debug single tests single run",
    "type": "node",
    "request": "launch",
    "env": { "CI": "true" },
    "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/jest",
    "args": ["--runInBand", "--no-cache", "--watchAll"],
    "cwd": "${workspaceRoot}",
    "program": "${fileBasenameNoExtension}",
    "protocol": "inspector",
    "console": "integratedTerminal",
    "internalConsoleOptions": "neverOpen"
}
```