Kenny R N Wilson

JavaScript

Characteristics and Benefits

Cheat Sheets

Basic Arrays

Name	Code	Result	Description
concat	`[1,2,3].concat([4,5,6])`	`[1,2,3,4,5,6])`	*Returns new array*
join	`[1,2,3].join(':')`	`1:2:3`	*Create a string from an array*
pop	`[1,2,3].pop()`	`3`	*Remove the last element and returns it*
shift	`[1,2,3].shift()`	`1`	*Remove and return the first element*
push	`[1,2,3].push(4)`	`Array now hold [1,2,3,4]`	*Append to end of array*
unshift	`[1,2,3].unshift(4)`	`Array now holds [4,1,2,3]`	
slice	`[1,2,3,4].slice(1,3)`	`[2,3]`	*Return a section of the array*
sort	`[4,2,3,5].sort()`	`[2,3,4,5]`	*Sort in place. Does not create new array*
splice	`var a = [1,2,3,4];` `var b = a.splice(1,2);`	`a=[1,4]` `b=[2,3]`	*Removes elements from array and*

LINQ

Name	Code	Result	Desc
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((prev,curr)=>curr+prev,0)`	`3`	

Spread

Name	Code

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

Use elements of array as Func args	`var numbers = [10,20,30]` `console.log(avg(...numbers));`
Use array to provide args	`var numbers = [10,20,30]` `console.log(avg(...numbers));`

Iterators

Name	Code

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

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

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

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));
```

Functions

Name	Code

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

Expression	`var f =  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 args	`var f =  () => 3.0`
Lambda 1 Arg	`var f = x => x * x ;`
Lambda 2 Args	`Var x = (x,y) => x + y;`

For in

Loop over elements in an array

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

Objects

Name	Code
Create Prototype	`function Car() {}`
Add Prototype Method	`Car.prototype.drive = function() {console.log("Broom!");}`
Add Property	`Object.defineProperty(Car.prototype,"length", { get: function() {return this._length;}, set: function(length) {this._length=length;} });`

The query selector methods take any valid CSS selector including pseudo selector. It is very 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*

The Type System

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

Operators

!! Convert to bool

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

|| to provide default value

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

Variables and Scope

Scope defines the visibility of variables. Scopes can be nested. A variable 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. JavaScript provides the var and let modifiers to specify the scope of variables.

VAR

Lexical Scope

Variable defined with **`var`** have lexical scope. Lexical scope means a variable is scoped by its execution context. When a variable is declared with var inside a function the execution context is the enclosing function. If the variable is declared with var outside of all functions, then its execution context is global.

```
// Global execution context
var a = 5;

function f()
{
    // Execution context of enclosing function
    var b = 15;
}

f();

console.log(a)

// Reference error.
console.log(b);
```

Declared and undeclared variables

JavaScript has the concept of declared and undeclared variables. A declared variable is declared with the var keyword and can take an optional initial value. An undeclared value has no var keyword and just takes an initial value. Undeclared variables implicitly take the global execution context

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

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

f();

console.log(a)

>> 10
```

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
```

Declared variables exist before any code in a file is executed.

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

a = 5;
console.log(a);

>> undefined
>> 5
```

Declared variables are a property of their execution context (global or function). As such they cannot be deleted. Undeclared variables can be deleted.

```
a = 5;

console.log(a);

delete a;

console.log(a);

>> 5
>> ReferenceError: a is not defined
```

UNDECLARED VARIABLES

> It is unwise to use undeclared variables and indeed in strict mode assigning to an undeclared variable is an error.

Hoisting

Variable declarations are processed before any code is executed. It is as if the variables were declared at the top of the file. This is known as hoisting.

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

var a;

>> 10
```

Hoisting does not affect the initialization. It occurs at the point the assignment statement is reached.

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
```

LET

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
```

Unlike var, variables declared with let at global scope do 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

Variables declared with let are not initialized until their definition is evaluated

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

    var a =5;
    let b = 6;
}

f();
>> ReferenceError: Cannot access 'b' before initialization
```

Similarly

```
function f()
{
    console.log(typeof b);
    let b = 6;
}

f();

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

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.

If we assign a value to an unassigned variable, then it is implicitly added to the global execution context. Note this would be an error in strict mode.

```
function f()
{
    a = 10;
    var b = 15;
}

f();

console.log(a);

>> 10
```

containing function. Code blocks have no impact. In the following code fragment the function `inner` can see any variables declared with var in its own body or in any enclosing functions.

```
function outer()
{
    var a = 5;

    return function inner()
    {
        console.log(a); // ok can see parent scope variables
    };
}
outer()();
// > 5
```

Closures

A closure is a combination of a function and its enclosing state. Functions are first-class objects. A function can reference any variable from its enclosing scope. If that function is passed around and used in another scope it will retain access to the original variables.

```
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
```

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
```

Functions

Functions are declared with the function keyword. They are first class objects and as such can be assigned to variables and passed as arguments to functions.

```
var f = function(x) {return x*x};
```

If we use the following special form the function can be called earlier in the file than its definition.

```
console.log(f2(9));
function f2(x) {return x*x};
```

Variable parameter lists are supported via the rest parameter syntax.

```
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 operator. The following uses the spread operator 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]));
```

Objects

TABLE 1OBJECT PROPERTIES

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

Methods	`var a = {`
	`  f: () => console.log("Hello")`
	`}`

Complex Properties	`var o3 = {`
	`  Name :`
	`  {`
	`  FirsName: "Kenny",`
	`  SecondName: "Wilson",`
	`  },`
	
	`  Age: 44`
	`}`

Dot operator	`console.log(o3.Name.FirsName);`
Keys	`var key = "Name";` `console.log(o3[key]["SecondName"]);`

Non-Enumerable Properties	`Object.defineProperty(Object.prototype,`
	`  "notshown",`
	`  {enumerable:false,value:"good"});`

A JavaScript object is essentially a collection of properties where each property associates a key with a value. The following defines an object using **literal format**

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

We can also create objects as follows (literal format is preferred)

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

We can use a variable to initialise a property of an object. The property name is the variable name and the value is the variable value.

```
var x = 2;
var o2 = {x};
console.log(o2);

>> {x:2}
```

Properties can be complex.

```
var o3 = {
    Name :
    {
        FirsName: "Kenny",
        SecondName: "Wilson",
    },

    Age: 44
}
```

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);

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

METHODS

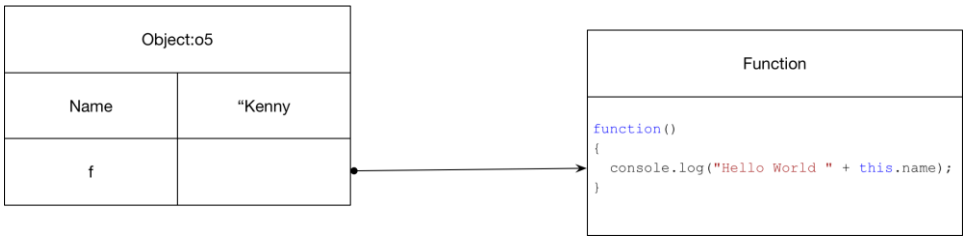
We can add functions to objects because functions are first class objects

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

  o5.print();

  >> Hello World
```

THIS AND THE EXECUTION CONTEXT

In JavaScript this 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 becomes the global context.

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

  var f = o5.print;
  f();

  >> Hello World undefined
```

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
```

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

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

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


>> Hello World John
```

PROTOTYPES

Create Prototype

```
function Car() {}
```

Add prototype method

```
Car.prototype.drive = function()
{console.log("Broom!");}
```

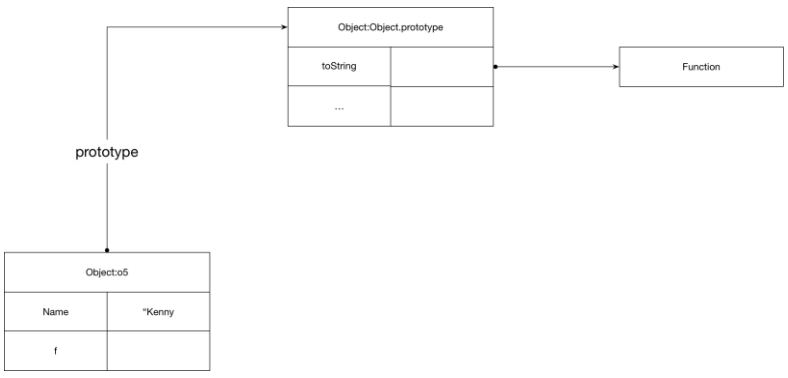
Add Property

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

The Default Prototype

JavaScript objects have prototypes. By default, all objects have Object.prototype as a prototype. The default prototype Object.prototype defines a basic `toString` method.

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

console.log(o5.toString());

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

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



If we want an object with no prototype, we can do as follows.

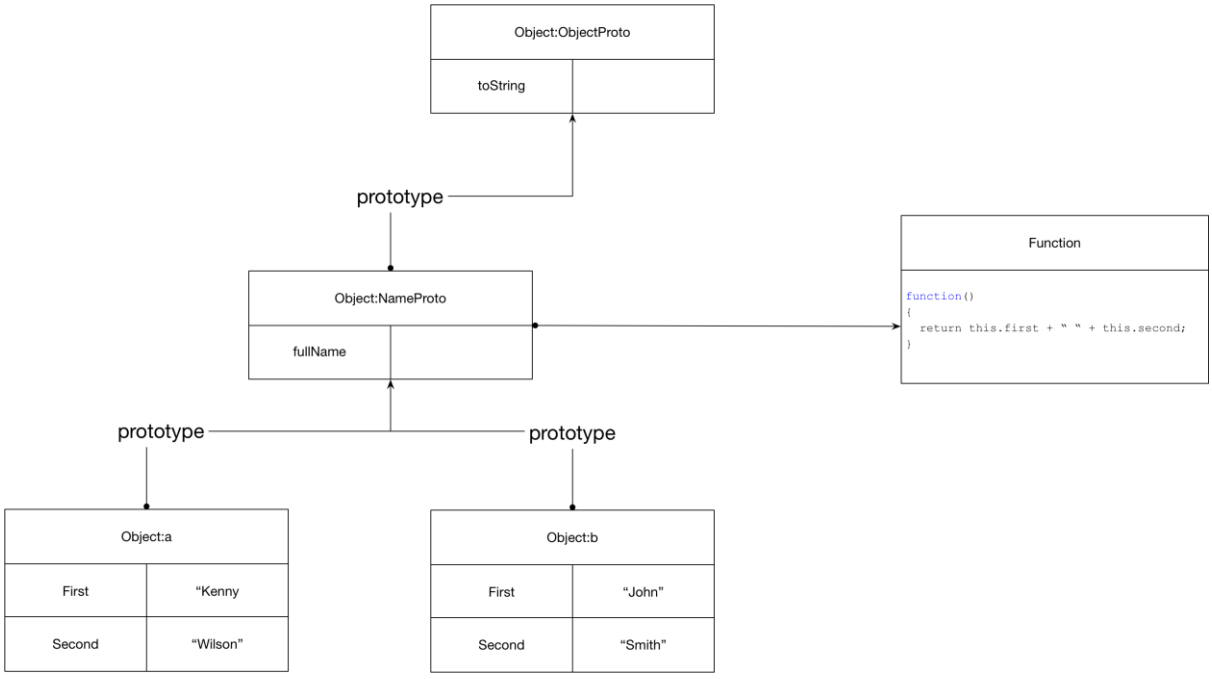
```
var pl =Object.create(null);
console.log(Object.getPrototypeOf(pl));
```

The prototype of Object.prototype is null

Creating prototype

The following code fragment shows how to create a common prototype. Note the importance of the execution context this. Each object uses provides its own execution context `this` so the prototype object executes on the correct context

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

var NameProto = {
    fullName: function() {x
        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
```



Constructors

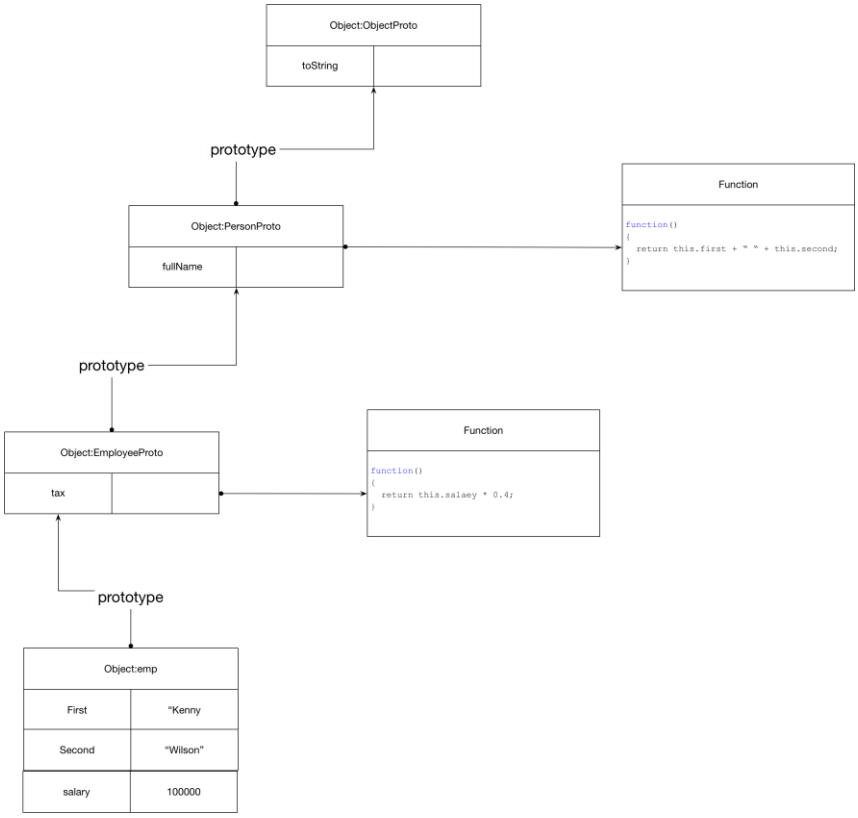
We can simplify the creation of objects using constructor functions. Constructor functions create a new object, initialise its properties and assign the new objects prototype. The following achieves the same as the previous section with less code

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

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

var name = new Name('Kenny', 'Wilson');
console.log(name.fullName());

>> Kenny Wilson
>> Sanna Wilson
```

Chaining Constructors

We need to be careful when chaining constructors

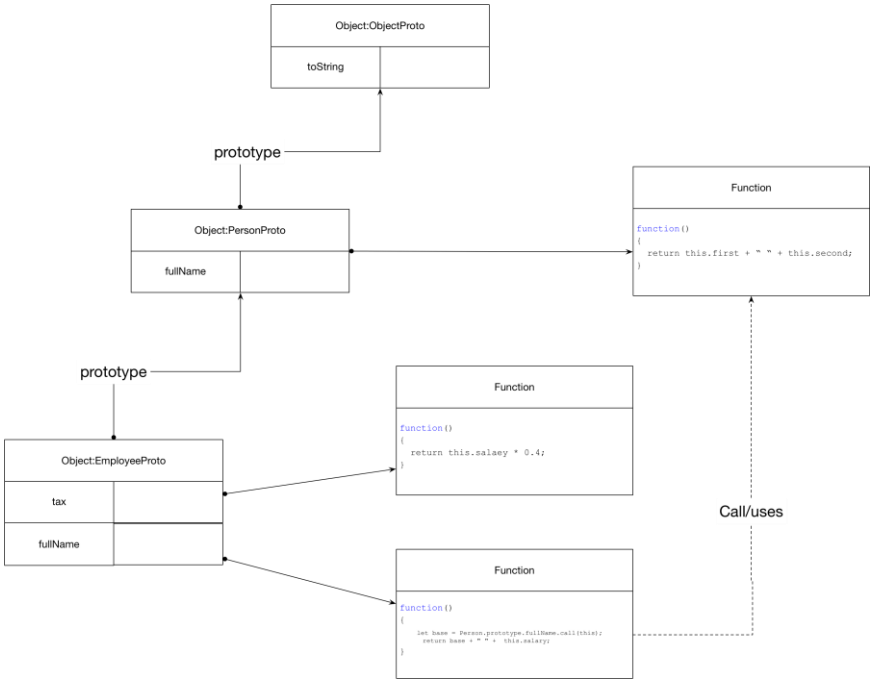
```
let Person = function(first,second)
{
    this.firstName = first;
    this.secondName = second;
}

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
```



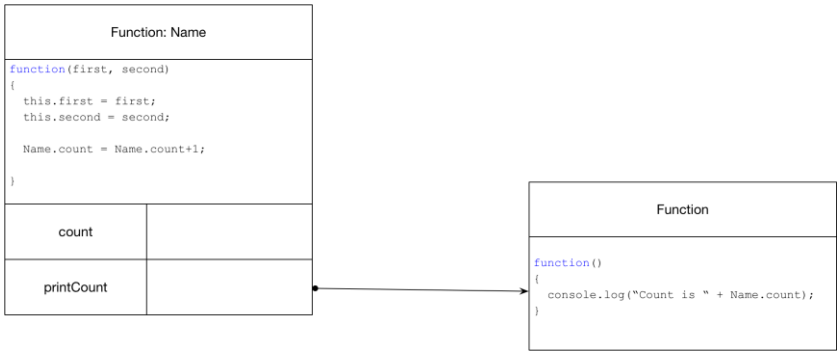
Instanceof

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

>> true
>> true
```

Static Properties and methods

The following code shows how to create a static method. Essentially as a method on the constructor 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);
```

Iterators, Iterables and Generators

ITERATOR

An iterator is an object that contains a method called next that returns objects with two properties as follows. 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());

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

ITERABLE

An iterable is an object that can produce an iterator. The following shows the relationship between an iterator and an iterable. Not how the spread operator can consume an iterable.

```
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));
```

GENERATING ITERABLE

The Language has support for generating Iterable.

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

    this[Symbol.iterator] = function* () {

        while (this.count >0)
        {
            yield this.count;
            this.count = this.count-1;
        }
    }
}
```

ITERABLE TYPES

The following language types are all iterable

- String
- Array
- TypedArray
- Map
- Set

CONSUMING SYNTAX

For Of

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

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

Spread

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

>> 1 2 3
```

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
```

Exceptions

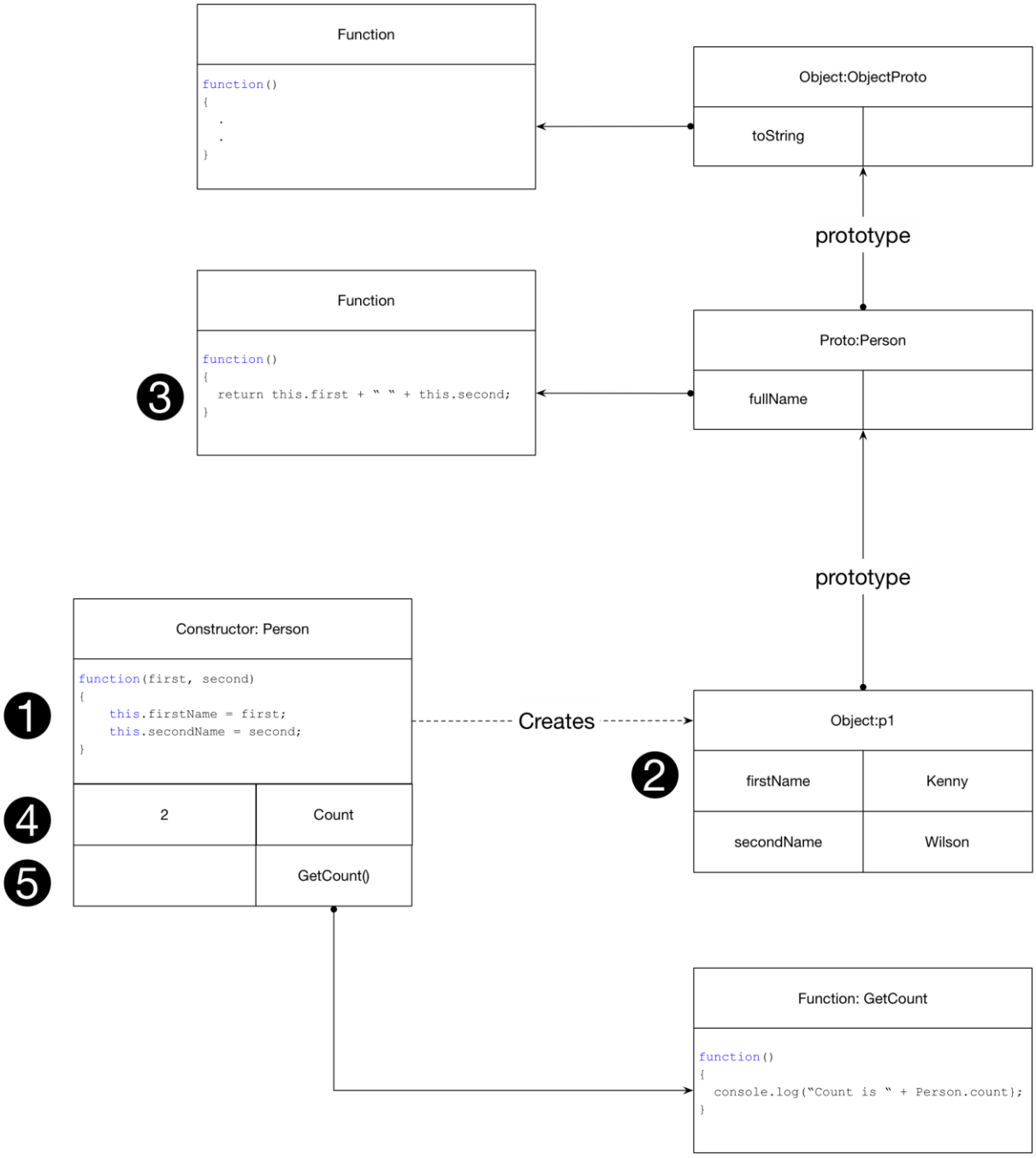
```
throw {message: 'a'};
```

Questions – The Type System

Objects

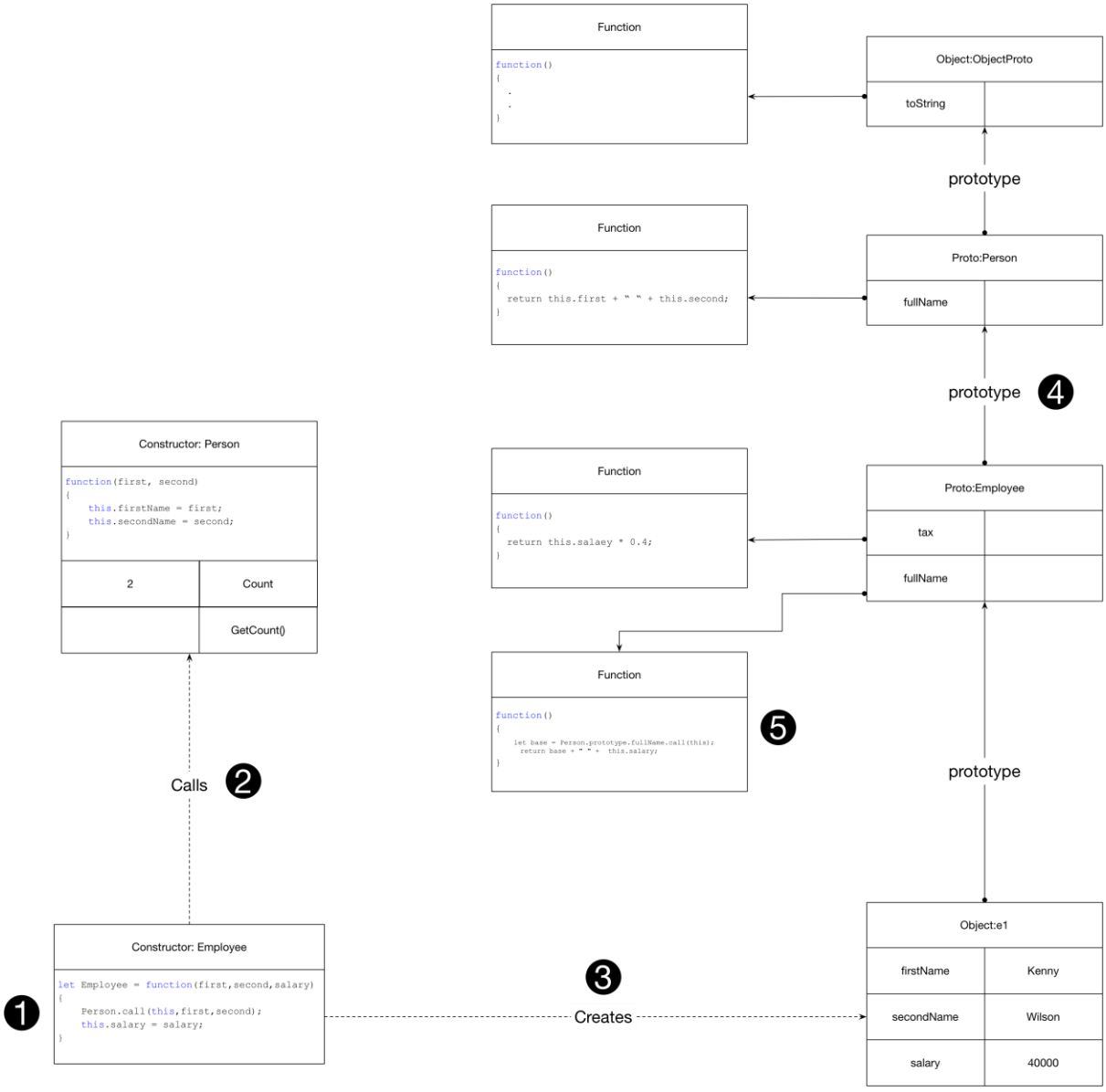
Basics

```
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");
```


Function
function()
{
.
.
}
Object:ObjectProto
toString
prototype
Function
function()
{
return this.first + " " + this.second;
}
3
Proto:Person
fullName
prototype
Constructor: Person
function(first, second)
{
this.firstName = first;
this.secondName = second;
}
1
Creates
Object:p1
2
firstName
Kenny
secondName
Wilson
4
2
Count
5
GetCount()
Function: GetCount
function()
{
console.log("Count is " + Person.count};
}

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

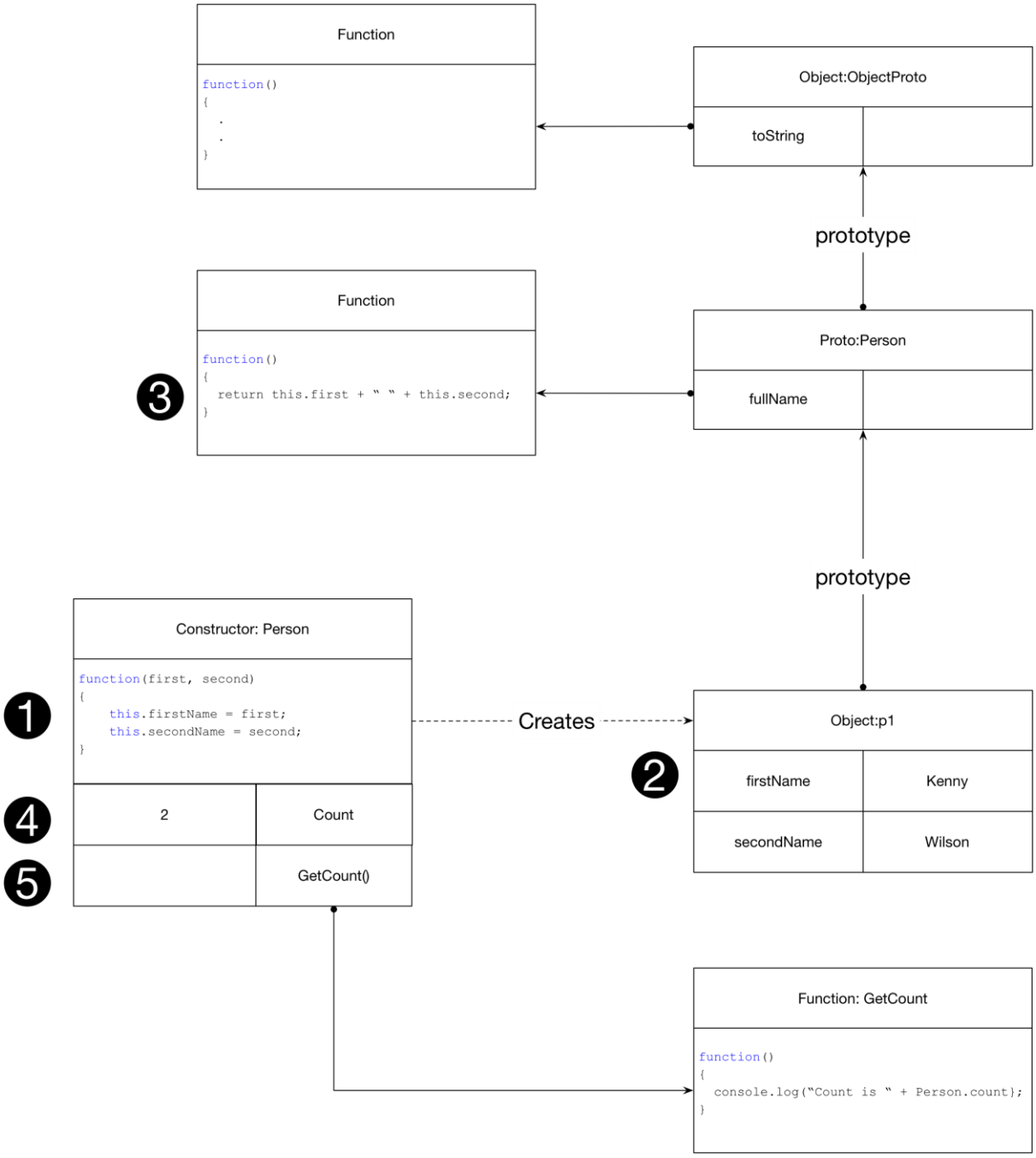
Classes

Basics

PROTOTYPES

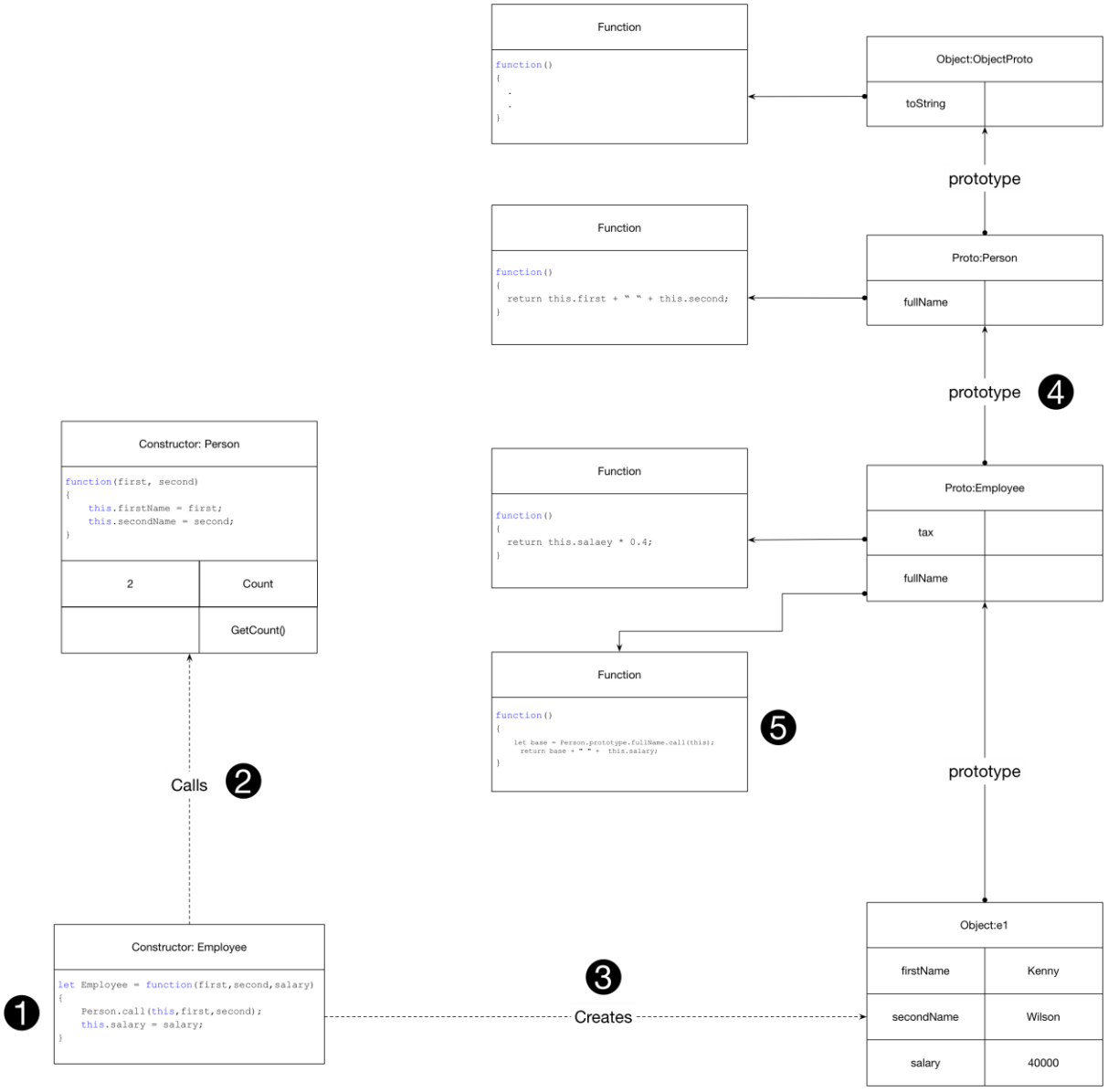
Classes are implemented using prototypes.

```
class Person
{
    ❶ constructor(first, second)
    {
        this.firstName = first;
        this.secondName = second;

        ++Person.Count ;
    }

   ❸ fullName() {
        return this.firstName + " " + this.secondName;
    }

    ❹ static Count = 0;

    ❺ static GetCount() {
        return Person.Count;
    }
}

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


Function
function()
{
.
.
}
Object:ObjectProto
toString
prototype
Function
function()
{
return this.first + " " + this.second;
}
3
Proto:Person
fullName
prototype
Constructor: Person
function(first, second)
{
this.firstName = first;
this.secondName = second;
}
1
Creates
Object:p1
2
firstName
Kenny
secondName
Wilson
4
2
Count
5
GetCount()
Function: GetCount
function()
{
console.log("Count is " + Person.count};
}

Inheritance

```
class Person
{
    constructor(first, second)
    {
        this.firstName = first;
        this.secondName = second;

        ++Person.Count ;
    }

    fullName() {
        return this.firstName + " " + this.secondName;
    }

    static Count = 0;

    static GetCount() {
        return Person.Count;
    }
}

class Employee extends  ❹ Person {

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

    tax() {
        return this.salary * 0.4;
    }

   ❺ fullName() {
        let base = super.fullName();
        return base + " " +  this.salary;
    }
}

❸
```


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

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())); } }`

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()));
    }
}

let kenny = new Employee('Kenny', 'Wilson', 100000);
let sanna = new Employee('Sanna', 'Hulkki', 40000);
Employee.printAll(kenny,sanna);

>>  Kenny Wilson 100000
>>  Sanna Hulkki 40000
```

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));
```

Modules

JavaScript modules have changed a lot down the years

ES5

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

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

export default myModule;
```

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 3 es5mod.js

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

module.exports = myModule;
```

Listing 4 modconsumer.js

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

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

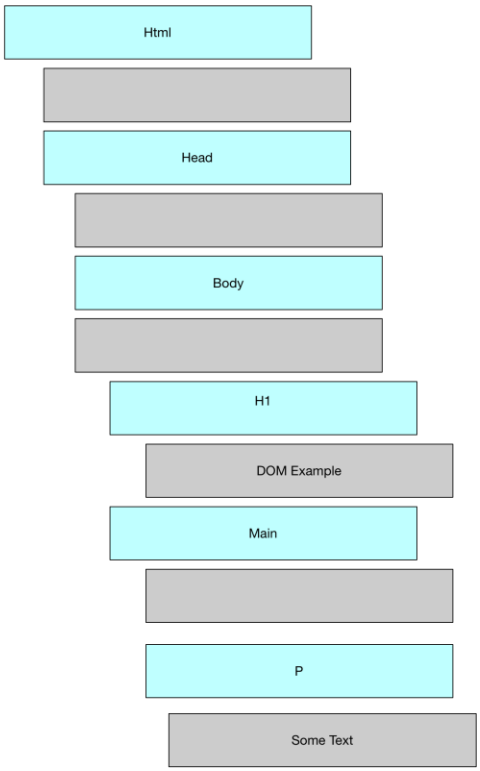
ES6

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

export default myModule;
```

The Browser

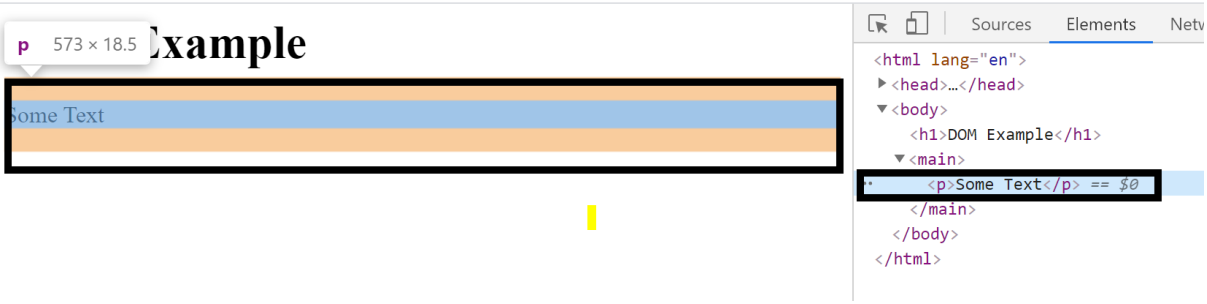
The DOM

```
<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.

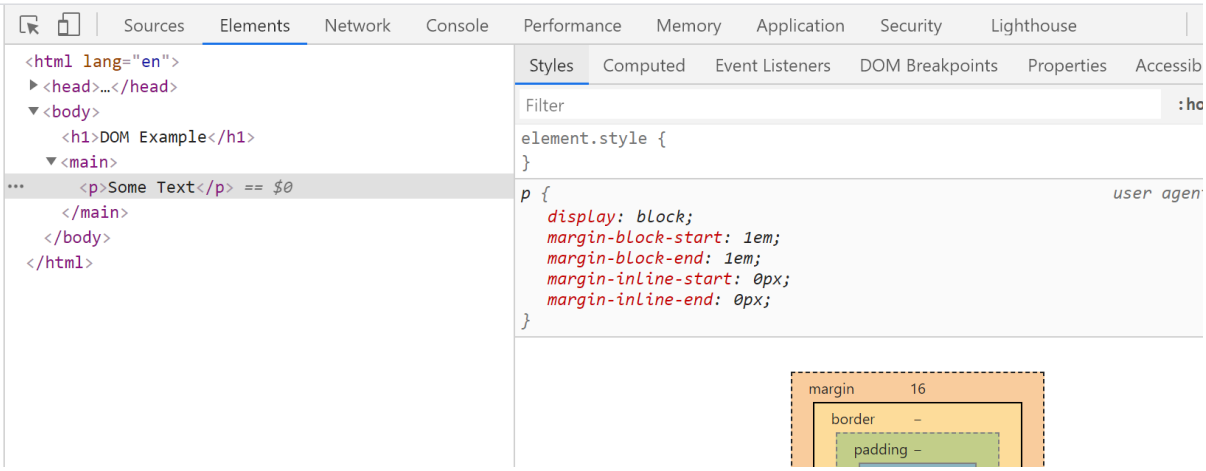
```
> document
  ▼#document
      <html lang="en">
      ▼<head>
          <title>Document</title>
        </head>
      ▼<body>
          <h1>DOM Example</h1>
        ▼<main>
            <p>Some Text</p>
          </main>
        </body>
      </html>
>
```

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 Core

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 second form allows the function to be used become it definition in the file. This is because it is treated as though the definition is at the beginning of the file.

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.

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"
}
```