**Closure Scope**

When a function is created, an invisible object is also created, this is known as the closure scope. Parameters and variables created in the function are stored on this invisible object.

When a function is inside another function, it can access both its own closure scope, and the parent closure scope of the outer function:

**function outerFn () {  
  var foo = true  
  function print() { console.log(foo) }  
  print() // prints true  
  foo = false  
  print() // prints false  
}  
outerFn()**

The outer variable is accessed when the inner function is invoked, this is why the second **print** call outputs **false** after **foo** is updated to **false**.

If there is naming collision then the reference to nearest close scope takes precedence:

**function outerFn () {  
  var foo = true  
  function print(foo) { console.log(foo) }  
  print(1) // prints 1  
  foo = false  
  print(2) // prints 2  
}  
outerFn()**

In this case the **foo** parameter of **print** overrides the **foo** var in the **outerFn** function.

Closure scope cannot be accessed outside of a function:

**function outerFn () {  
  var foo = true  
}  
outerFn()  
console.log(foo) // will throw a ReferenceError**

Since the invisible closure scope object cannot be accessed outside of a function, if a function returns a function the returned function can provide controlled access to the parent closure scope. In essence, this provides encapsulation of private state:

**function init (type) {  
  var id = 0  
  return (name) => {  
    id += 1  
    return { id: id, type: type, name: name }  
  }  
}  
const createUser = init('user')  
const createBook = init('book')  
const dave = createUser('Dave')  
const annie = createUser('Annie')  
const ncb = createBook('Node Cookbook')  
console.log(dave) //prints {id: 1, type: 'user', name: 'Dave'}  
console.log(annie) //prints {id: 2, type: 'user', name: 'Annie'}  
console.log(ncb) //prints {id: 1, type: 'book', name: 'Node Cookbook'}**

The **init** function sets a variable **id** in its scope, takes an argument called **type**, and then returns a function. The returned function has access to **type** and **id** because it has access to the parent closure scope. Note that the returned function in this case is a fat arrow function. Closure scope rules apply in exactly the same way to fat arrow functions.

The **init** function is called twice, and the resulting function is assigned to **createUser** and **createBook**. These two functions have access to two separate instances of the **init** functions closure scope. The **dave** and **annie** objects are instantiated by calling **createUser**.

The first call to **createUser** returns an object with an **id** of **1**. The **id** variable is initialized as **0** and it is incremented by 1 before the object is created and returned. The second call to **createUser** returns an object with **id** of **2**. This is because the first call of **createUser** already incremented **id** from **0** to **1**, so on the next invocation of **createUser** the **id** is increased from **1** to **2**. The only call to the **createBook** function however, returns an **id** of **1** (as opposed to 3), because **createBook** function is a different instance of the function returned from **init** and therefore accesses a separate instance of the **init** function's scope.

In the example all the state is returned from the returned function, but this pattern can be used for much more than that. For instance, the **init** function could provide validation on **type**, return different functions depending on what **type** is.

Another example of encapsulating state using closure scope would be to enclose a secret:

**function createSigner (secret) {  
  const keypair = createKeypair(secret)  
  return function (content) {  
     return {  
        signed: cryptoSign(content, keypair.privateKey),  
        publicKey: keypair.publicKey  
     }  
  }  
}  
const sign = createSigner('super secret thing')  
const signedContent = sign('sign me')  
const moreSignedContent = sign('sign me as well')**

Note, in this code **createKeypair** and **cryptoSign** are imaginary functions, these are purely for outlining the concept of the encapsulation of secrets.

Closure scope can also be used as an alternative to prototypal inheritance. The following example provides equivalent functionality and the same level of composability as the three prototypal inheritance examples but it doesn't use a prototype chain, nor does it rely the implicit **this** keyword:

**function wolf (name) {  
  const howl = () => {  
    console.log(name + ': awoooooooo')  
  }  
  return { howl: howl }  
}**

**function dog (name) {  
  name = name + ' the dog'  
  const woof = () => { console.log(name + ': woof') }  
  return {  
    ...wolf(name),  
    woof: woof  
  }  
}  
const rufus = dog('Rufus')**

**rufus.woof() // prints "Rufus the dog: woof"  
rufus.howl() // prints "Rufus the dog: awoooooooo"**

The three dots (**...**) in the return statement of **dog** is called the spread operator. The spread operator copies the properties from the object it proceeds into the object being created.

The **wolf** function returns an object with a **howl** function assigned to it. That object is then spread (using …) into the object returned from the **dog** function, so **howl** is copied into the object. The object returned from the **dog** function also has a **woof** function assigned.

There is no prototype chain being set up here, the prototype of **rufus** is **Object.prototype** and that's it. The state (**name**) is contained in closure scope and not exposed on the instantiated object, it's encapsulated as private state.

The **dog** function takes a **name** parameter, and immediately reassigns it to **name + ' the dog'**. Inside **dog** a **woof** function is created, where it references **name**. The **woof** function is returned from the **dog** function inside of an object, as the **woof** property. So when **rufus.woof()** is called the **woof** accesses **name** from it's parent scope, that is, the closure scope of **dog**. The exact same thing happens in the **wolf** function. When **rufus.howl()** is called, the **howl** function accesses the **name** parameter in the scope of the **wolf** function.

The advantage of using closure scope to compose objects is it eliminates the complexity of prototypes, context (**this**) and the need to call a function with **new –** which when omitted can have unintended consequences. The downside is that where a prototype method is shared between multiple instances, an approach using closure scope requires that internal functions are created per instance. However, JavaScript engines use increasingly sophisticated optimization techniques internally, it's only important to be fast enough for any given use case and ergonomics and maintainability should take precedence over every changing performance characteristics in JavaScript engines. Therefore it's recommended to use function composition over prototypal inheritance and optimize at a later point if required.