|  |  |
| --- | --- |
| (1)var a = "global variable";  var F = function() {  var b = "local variable";  var N = function() {  var c = "inner local";  return b;  };  return N;  };  var inner = F();  var b = inner();  console.log(b); | (3)var x = "global variable";  (function() {  var y = "local variable";  var r = function() {  var z = "inner local";  return y;  //console.log(y);  };  })() |
| (2)function F() {  var arr = [],  i;  for (i = 0; i < 3; i++) {  arr[i] = function() {  var b = i;  console.log(b);  return i;  };  }  return arr;  }  var arr = F();  arr[0](); | (4)function f(x) {  function g() {  return x;  }  return g;  }  //Tell f to create a new g  var g5 = f(5);  //g5 is a function that always returns 5  alert(g5());  //Tell f to create another new g  var g1 = f(1);  //g1 is a function that always returns 1  alert(g1());  /\*\*\*\*\*\*\*Anonymous Function \*\*\*\*\*/  function f(x) {  return function() {  return x;  }  } |
| (5)function person(name) {  function get() {  return name;  }  function set(newName) {  name = newName;  }  return [get, set];  }  var getSetDave = person('Dave');  var getDave = getSetDave[0];  var setDave = getSetDave[1];  alert(getDave()); //'Dave'  setDave('Bob');  alert(getDave()); //'Bob'  var getSetMary = person('Mary');  var getMary = getSetMary[0];  var setMary = getSetMary[1];  alert(getMary()); //'Mary' |

|  |  |  |  |
| --- | --- | --- | --- |
| Clousre (Closures are functions that retain a reference to their free variables) | | Prototype | |
| function CatMaker(name) {  var age = 10;  //construct an object on the fly with three methods.  //All methods have access to age, but age cannot be  //directly accessed outside of this function.  return {  "sayHello": function () { //first method  alert("Miaow");  },  "getAge": function (inCatYears) { //second method  if (inCatYears) {  return age \* 7;  }  else {  return age;  }  },  "makeOlder": function () { //third method  age++;  }  };  }  var mycat = CatMaker('Snuffles');  mycat.getAge(true); //returns 70  mycat.makeOlder();  mycat.getAge(true); //returns 77 | | var TeslaModelS = function() {  this.numWheels = 4;  this.manufacturer = 'Tesla';  this.make = 'Model S';  }  TeslaModelS.prototype.go = function() {  // Rotate wheels  alert('go');  }  TeslaModelS.prototype.stop = function() {  // Apply brake pads  alert('stop');  }  var TelObj = new TeslaModelS();  TelObj.go(); | |
| **Prototype Design Pattern** |  | | **Revealing Prototype Pattern** | |
| var TeslaModelS = function() {  this.numWheels = 4;  this.manufacturer = 'Tesla';  this.make = 'Model S';  }  TeslaModelS.prototype.go = function() {  // Rotate wheels  alert('go');  }  TeslaModelS.prototype.stop = function() {  // Apply brake pads  alert('stop');  }  var TelObj = new TeslaModelS();  TelObj.go(); | var TeslaModelS = function() {  this.numWheels = 4;  this.manufacturer = 'Tesla';  this.make = 'Model S';  }  TeslaModelS.prototype = {  go: function() {  // Rotate wheels  alert('go');  },  stop: function() {  // Apply brake pads  alert('stop');  }  }  var TelObj = new TeslaModelS();  TelObj.go(); | | var TeslaModelS = function() {  this.numWheels = 4;  this.manufacturer = 'Tesla';  this.make = 'Model S';  }  TeslaModelS.prototype = function() {  var go = function() {  // Rotate wheels  alert('go');  };  var stop = function() {  // Apply brake pads  alert('stop');  };  return {  pressBrakePedal: stop,  pressGasPedal: go  }  }();  var TelObj = new TeslaModelS();  TelObj.make;  TelObj.pressBrakePedal(); | |

**Closures vs Objects :**

In JavaScript, there are two main patterns for creating objects with state - plain JavaScript objects and closures. An object is an entity with state and methods to access/modify that state. There are two main approaches to this, which I’ll be calling “objects” and “closures”.

|  |  |
| --- | --- |
| //object  function Person(name) {  this.name = name;  }  Person.prototype.sayHi = function () {  console.log('Hi there, my name is ' + this.name);  };  //closure  function person(name) {  //using object literal but state held in closure, not in object  return {  sayHi: function () {  console.log('Hi there, my name is ' + name);  }  };  } | //alternative object  function person(name) {  return {  name: name,  sayHi: function () {  console.log('Hi there, my name is ' + this.name);  }  };  } |

The point is, using objects, state is shared through **this**. Whenever you call a method on an object, e.g.

var dave = new Person('Dave');

dave.sayHi();

**this** within the method will be equal to the object it was called upon.

When using closures however, state is shared through the lexical scope. This highlights the first key difference between objects and closures: access to the internal state.

There are three ways to mutate the internal state of an object in JavaScript.

//Obtain a reference to the object and assign new properties

function changeName(object, newName) {

object.name = newName;

}

changeName(dave, 'Bob');

//Attach a function to the object and call it as a method on the object

function changeName(newName) {

this.name = newName;

}

dave.changeName = changeName;

dave.changeName('Bob');

//call/apply a function with the object as the context

changeName.call(dave, 'Bob');

With closures, on the other hand, there is only one way to mutate the internal state – be inside the scope of the closure:

function person(name) {

//to change `name`, you \*must\* be defined somewhere inside this function

return {

sayHi: function () {

console.log('Hi there, my name is ' + name);

}

};

}

1. The advantage of objects over closures is that you’re not limited in the functionality you can add to an object by location in the source code.
2. With a closure the only way to add functionality (with access to the internal state) is to define it somewhere inside the function that creates the closure.
3. From the perspective of third party code. With an object, anybody can add or change functionality on your object, and access its internal state. With a closure, the internal state is private – it can’t be accessed from outside the closure without the use of accessor functions.
4. Another advantage that objects have is in terms of memory usage. With a closure, by definition, for a function to have access to the internal state it must be *defined* inside the closure. That means that each new closure created *must* have its own version of the function. Objects on the other hand have no such limitation. A function that reads from and writes to this need only be defined once – it can then be added to any object, either shared via the prototype system or through other means.
5. An advantage that closures have is that you don’t need to keep track of this. With closures it’s simple – you’re either in the correct scope or you’re not. With objects, if you’re in a method called on an object then this is that object, but if the method gets detached from its object (e.g. when passed as an argument) it loses its binding, or if you have a nested function inside the method. Then you need to start messing about with call and apply, and bind and other fun stuff.
6. So, when should you use objects and when closures? If you’re making hundreds of object-type things then they should probably be objects. If there are only a few and you have security concerns then closures are a better bet.
7. Privacy isn’t just a security issue – it’s useful for creating a clean separation between the public API and the private implementation. Some developers would say that JavaScript doesn’t provide privacy, so you should get used to writing objects with everything public, and tell users of your code to just not touch certain properties (for example those with a leading underscore). I think it’s useful being able to enforce privacy – it makes sure that no-one will ever write code that’s reliant on an implementation detail of yours.
8. Because of this, I tend to favour closures, falling back to objects when memory becomes a concern, but that’s largely a personal preference.