```
Object.prototype.instanceof = function (prototype) {
      var object = this;

      do {
          if (object === prototype) return true;
          var object = Object.getPrototypeOf(object);
      } while (object);

      return false;
 };

function foo() {

}

var bar = { a:'a'};

foo.prototype = bar;

baz = Object.create(bar);

console.log(baz instanceof foo);

console.log(baz.instanceof(foo) );


const arr = [10,12,15,21];

for (var i=0;i<arr.length;i++) {

  setTimeout(function(_i) {

    console.log('index is '+ _i);

  }(i),3000);

}



Delegation / Differential Inheritance

var proto = {
  hello: function hello() {
    return 'Hello, my name is ' + this.name;
  },
  habit: ['sleep','eat']
};

var george = Object.create(proto);
george.name = 'George';
```

```
var george1 = Object.create(proto);
george1.name = 'George';

console.log(george);
console.log(george1);
// The one major drawback to delegation is that it's not very good at storing
state. In particular, if you try to store state as objects or arrays,
mutating any member of the object or array will mutate the member for every
instance that shares the prototype.
//george1.habit[0] = 'coding';
//george1.name ='2';
//console.log(george);
//console.log(george1);

~~function Greeter(name) {~~
~~  this.name = name || 'John Doe';~~
~~}~~
~~Greeter.prototype.hello = function hello() {~~
~~  return 'Hello, my name is ' + this.name;~~
~~}~~
~~var george = new Greeter('George');~~


Cloning / Concatenative Inheritance / Mixins

var proto = {
  hello: function hello() {
    return 'Hello, my name is ' + this.name;
  }
};

//var george = _.extend({}, proto, {name: 'George'});
var george = Object.assign({}, proto, {name: 'George'});

george;


Closure Prototypes / Functional Inheritance

Closure prototypes are functions that can be run against a target object
in order to extend it. The primary advantage of this style is that it allows
for encapsulation. In other words, you can enforce private state.

function base(spec) {
    var that = {}; // Create an empty object
    that.name = spec.name; // Add it a "name" property

    var age = 10;
    that.getAge = function() {return age;}

    return that; // Return the object
}
```

```
// Construct a child object, inheriting from "base"
function child(spec) {
    var that = base(spec); // Create the object through the "base"
constructor
    that.sayHello = function() { // Augment that object
        return 'Hello, I\'m ' + that.name;
    };
    return that; // Return it
}

// Usage
var object1 = child({ name: 'a functional object' });

console.log( object1 );
console.log( object1.getAge() );

//------------------------------
var george ={} ;

var model = function (v1) {

  this.attrs = {'attr1':v1};

  this.set = function (name, value) {
    this.attrs[name] = value;
    this[name] = value;
    //this.trigger('change', {name: name, value: value });
  };

  this.get = function (name, value) {
    return this.attrs[name];
  };

  //_.extend(this, Backbone.Events);
};

model.call(george, 'Top secret');

//george.on('change', function (e) { console.log(e); });

george.set('name', 'Sam');

george;
```