JavaScript Advanced

- Scope
- IIFE and Closure
- Asynchronous
- Constructor
- Prototype
- Inheritance
- ES6 features

Scope

A variable declared outside a function, is a global variable.

- A global variable has **global scope**: All scripts and functions on a web page can access it.

Local

- Variables declared within a JavaScript function, become **LOCAL** to the function.
- Local variables have **Function scope**: They can only be accessed from within the function.

Closure

- *Closures* are simply functions that access data outside their own scope.

```
function fn1() {
    var b=2;
    var a= 1;
     function fn2() {
          console.log(a);
      }
    return fn2;
  }

  var result = fn1();
  result(); // 1
```

Closure

```
var person = (function() {

    var age = 25;

    return {
        name: "Nicholas",

        getAge: function() {
            return age;
        },

        growOlder: function() {
            age++;
        }
    };

}());

console.log(person.name);       // "Nicholas"
console.log(person.getAge());   // 25

person.age = 100;
console.log(person.getAge());   // 25

person.growOlder();
console.log(person.getAge());   // 26
```

IIFE

An IIFE is a function expression that is defined and then called immediately to produce a result.

That function expression can contain any number of local variables that aren’t accessible from outside that function.

IIFE

```
function add() {
    var counter = 0;
    function plus() {counter += 1;}
    plus();
    return counter;
}
```

```
var add = (function () {
    var counter = 0;
    return function () {counter += 1; return counter}
})();

add();
add();
add();
```

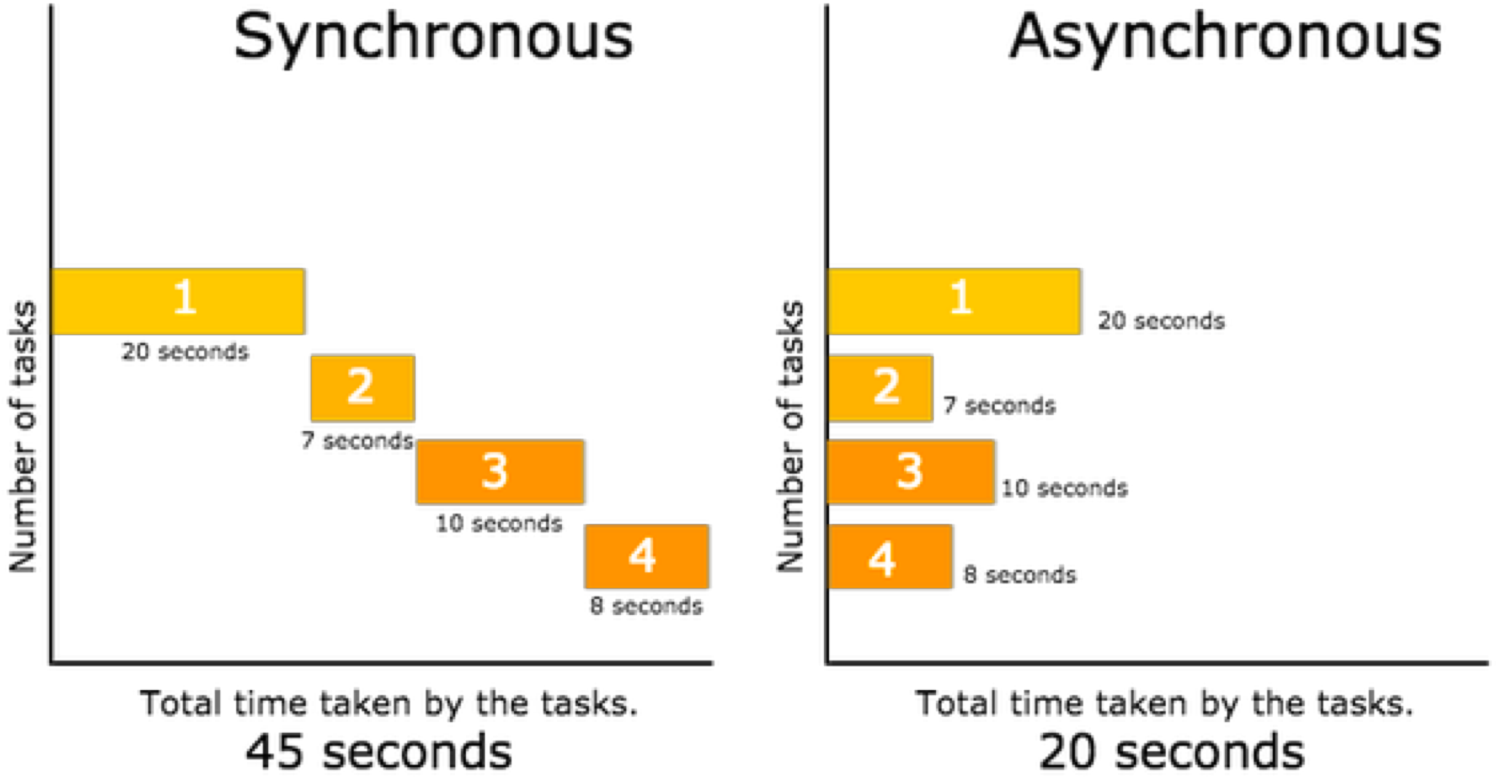
Question

```
var num = 10;
var obj = {
  num:8,
  inner: {
    num:6,
    print1:function () {
      console.log(this.num);
    },
    print2: () => {
      console.log(this.num)
    }

  }
}

num = 888;
obj.inner.print1();
var fn = obj.inner.print1;
fn();
(obj.inner.print1)();
(obj.inner.print2)();
(obj.inner.print = obj.inner.print1)()
```

Asynchronous

- In *asynchronous* programs, you can have two lines of code (L1 followed by L2), where L1 schedules some task to be run in the future, but L2 runs before that task completes.

Synchronous VS Asynchronous



Handle Asynchronized Problem

Callback function

- function doHomework(subject, callback) {
  alert(`Starting my ${subject} homework.`);
  callback();
  }
- function alertFinished(){
  alert('Finished my homework');
  }
- **doHomework('math', alertFinished);**

Promise

Observable

Example

```
for(var i = 0; i < 10; i++) {
    setTimeout(function() {
        console.log('i is ' + i);
    }, 1000);
}
```

```
for(var i = 0; i < 10; i++) {
    print(i);
}
function print(str) {
    setTimeout(function() {
        console.log('i is ' + str);
    }, 1000);
}
```

Constructor

A *constructor* is simply a function that is used with *new* to create an object.

The advantage of constructors is that objects created with the same constructor contain the same properties and methods.

Because a constructor is just a function, you define it in the same way. The difference is that constructor names should begin with a capital letter, to distinguish them from other functions.

Constructor

```
function Person() {
    // intentionally empty
}

var person1 = new Person();
var person2 = new Person();

var person1 = new Person;
var person2 = new Person;
```

```
console.log(person1 instanceof Person);      // true
console.log(person2 instanceof Person);      // true

console.log(person1.constructor === Person);     // true
console.log(person2.constructor === Person);     // true
```

Constructor

- A *constructor* with parameter, properties and methods:

```
function Person(name) {
    this.name = name;
    this.sayName = function() {
        console.log(this.name);
    };
}
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1.name);          // "Nicholas"
console.log(person2.name);          // "Greg"

person1.sayName();                  // outputs "Nicholas"
person2.sayName();                  // outputs "Greg"
```

Constructor

- When calling a *constructor* without *new* keyword

```
var person1 = Person("Nicholas");          // note: missing "new"

console.log(person1 instanceof Person);    // false
console.log(typeof person1);               // "undefined"
console.log(name);                         // "Nicholas"
```

- When *Person* is called as a function without *new*, the value of *this* inside of the constructor is equal to the global *this* object. The variable *person1* doesn't contain a value because the *Person* constructor relies on *new* to supply a return value. Without *new*, *Person* is just a function without a return statement. The assignment to *this.name* actually creates a global variable called *name*, which is where the *name* passed to *Person* is stored.

Constructors

Constructors allow you to configure object instances with the same properties, but constructors alone don't eliminate code redundancy

In the example code thus far, each instance has had its own *sayName()* method even though *sayName()* doesn't change. That means if you have 100 instances of an object, then there are 100 copies of a function that do the exact same thing, just with different data.

It would be much more efficient if all of the instances shared one method, and then that method could use *this.name* to retrieve the appropriate data. This is where *prototypes* come in.

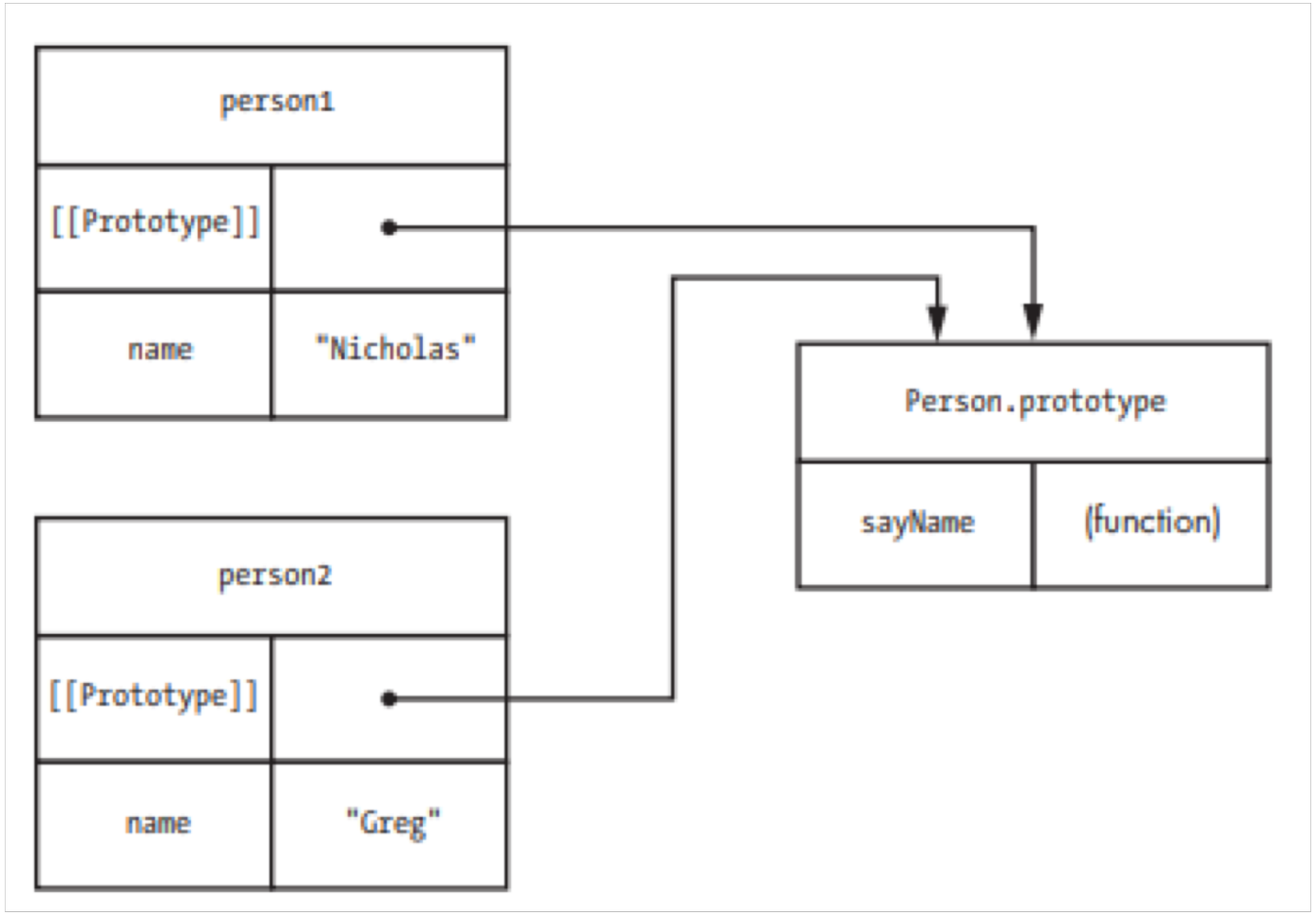
Prototype

- Almost every function (with the exception of some built-in functions) has a *prototype* property that is used during the creation of new instances. That prototype is shared among all of the object instances, and those instances can access properties of the prototype.

```
var book = {
    title: "The Principles of Object-Oriented JavaScript"
};

console.log("title" in book);                                    // true
console.log(book.hasOwnProperty("title"));                       // true
console.log("hasOwnProperty" in book);                           // true
console.log(book.hasOwnProperty("hasOwnProperty"));              // false
console.log(Object.prototype.hasOwnProperty("hasOwnProperty"));  // true
```

[[Prototype]] Property

- An instance keeps track of its prototype through an internal property called [[*Prototype*]].
- This property is a pointer back to the prototype object that the instance is using.
- When you create a new object using new, the constructor's prototype property is assigned to the [[Prototype]] property of that new object.



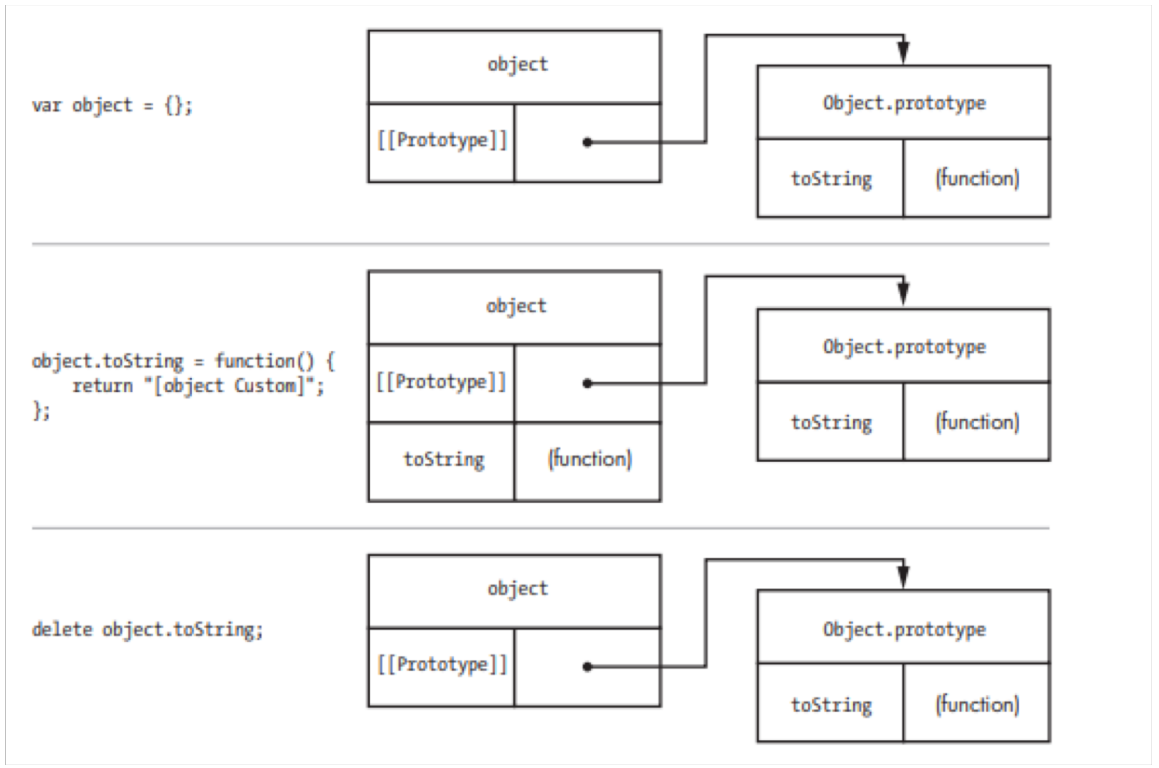
Read *[[Prototype]]* Property

- You can read the value of the [[*Prototype*]] property by using the Object.getPrototypeOf() method on an object.

```
var object = {};
var prototype = Object.getPrototypeOf(object);

console.log(prototype === Object.prototype);            // true
```

How [[Prototype]] Property read/write works

```
var object = {};

console.log(object.toString());    // "[object Object]"

object.toString = function() {
    return "[object Custom]";
};

console.log(object.toString());    // "[object Custom]"

// delete own property
delete object.toString;
console.log(object.toString());    // "[object Object]"

// no effect - delete only works on own properties
```



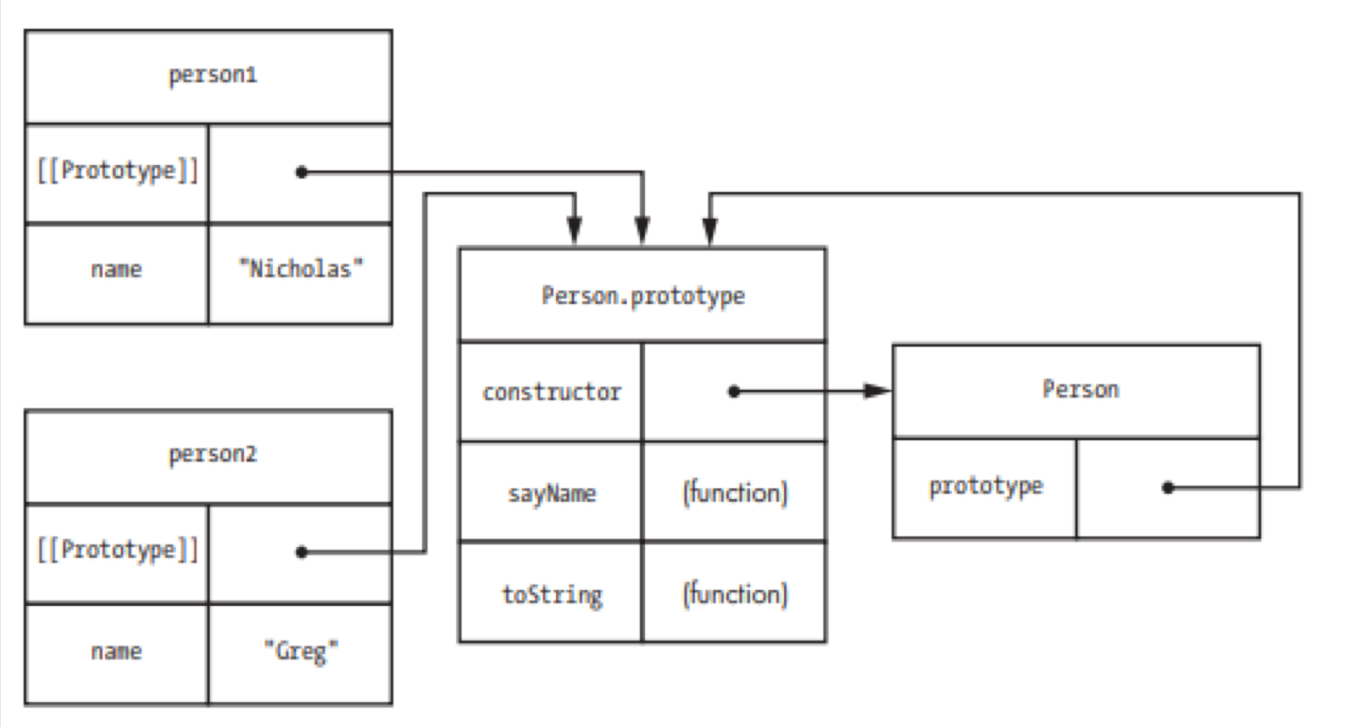
Prototype with Constructors

- The shared nature of prototypes makes them ideal for defining methods once for all objects of a given type.
- It's much more efficient to put the methods on the prototype and then use this to access the current instance.

```
function Person(name) {
    this.name = name;
}

Person.prototype.sayName = function() {
    console.log(this.name);
};

var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1.name);          // "Nicholas"
console.log(person2.name);          // "Greg"

person1.sayName();                  // outputs "Nicholas"
person2.sayName();                  // outputs "Greg"
```

Prototype with Constructors

- The relationships among constructors, prototypes, and instances is that there is no direct link between the instance and the constructor. There is, however, a direct link between the instance and the prototype and between the prototype and the constructor



Replacing prototype with object literal

```
function Person(name) {
    this.name = name;
}

Person.prototype = {
    constructor: Person,

    sayName: function() {
        console.log(this.name);
    },

    toString: function() {
        return "[Person " + this.name + "]";
    }
};

var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1 instanceof Person);          // true
console.log(person1.constructor === Person);     // true
console.log(person1.constructor === Object);     // false

console.log(person2 instanceof Person);          // true
console.log(person2.constructor === Person);     // true
console.log(person2.constructor === Object);     // false
```

Changing Prototypes

```
function Person(name) {
    this.name = name;
}

Person.prototype = {
    constructor: Person,

    sayName: function() {
        console.log(this.name);
    },

    toString: function() {
        return "[Person " + this.name + "]";
    }
};
```

```
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log("sayHi" in person1);        // false
console.log("sayHi" in person2);        // false

// add a new method
Person.prototype.sayHi = function() {
    console.log("Hi");
};

person1.sayHi();                        // outputs "Hi"
person2.sayHi();                        // outputs "Hi"
```

JS Inheritance

- When you attempt to access a property or method of an object, JavaScript will first search on the object itself, and if it is not found, it will search the object's [[Prototype]]. If after consulting both the object and its [[Prototype]] still no match is found, JavaScript will check the prototype of the linked object, and continue searching until the end of the prototype chain is reached.

JS Inheritance

Prototype Inheritance

Object Inheritance

Prototype Inheritance

```
function Dog (name) {
  this.name = name;
}

Dog.prototype.species = 'dog';

var dogA = new Dog ('Monday');
var dogB = new Dog ('Sunday');

console.log(dogA);
console.log(dogA.species);          //dog
console.log(dogB.species);          //dog


Dog.prototype.species = 'cat';
console.log(dogA.species);          //cat
console.log(dogB.species);          //cat
```

```
function Dog (name) {
  this.name = name;
  this.species = 'dog'
}

var dogA = new Dog('Monday');
var dogB = new Dog('Sunday');

console.log(dogA.name);       //Monday
console.log(dogB.name);       //Sunday

dogA.species = 'cat';
console.log(dogA.species);    //cat
console.log(dogB.species);    //dog
```

Object Inheritance

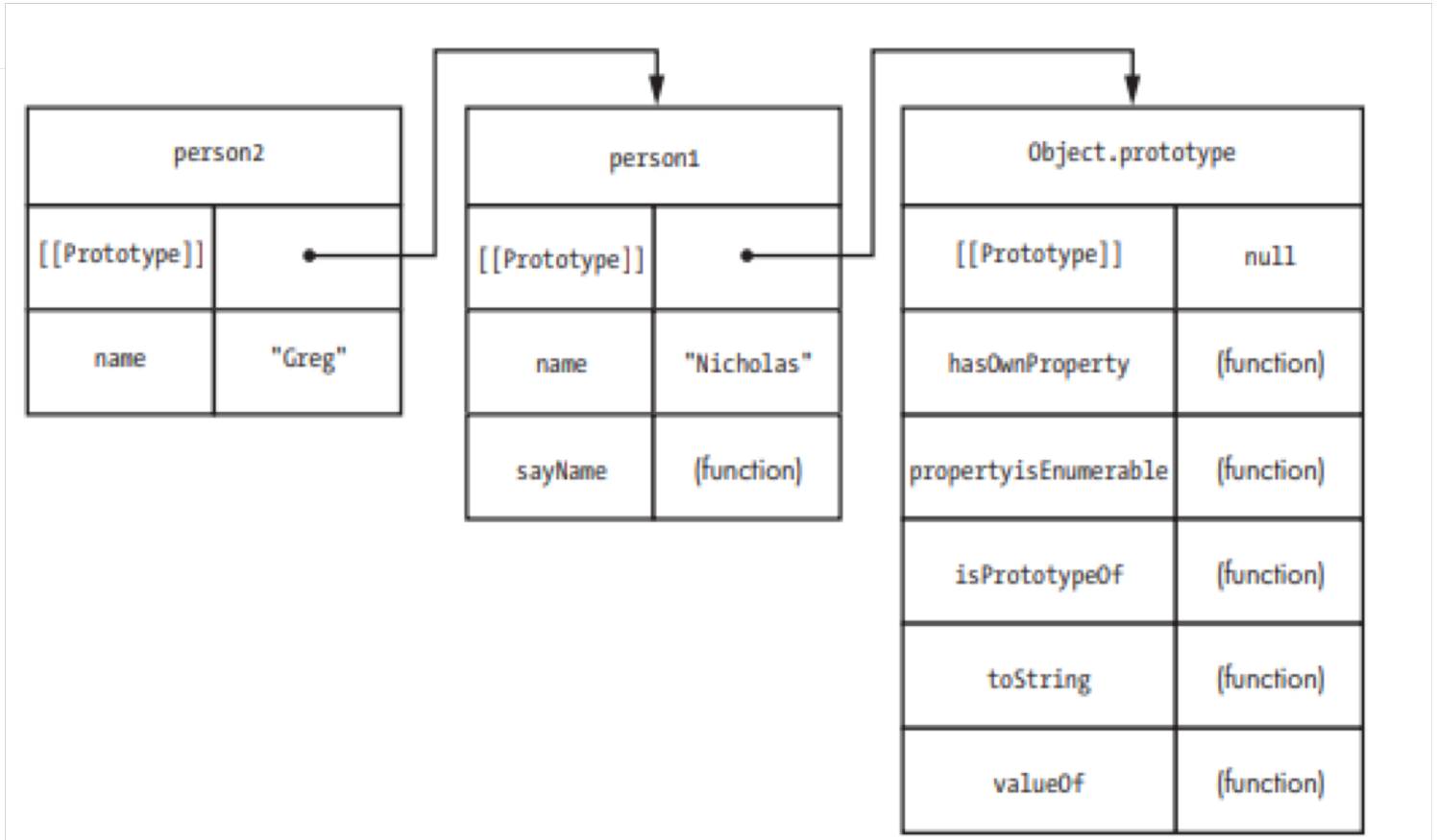
Specify what object should be the new object's [[Prototype]].

- Object literals have Object.prototype set as their [[Prototype]] implicitly, but you can also explicitly specify [[Prototype]] with the Object.create() method.

```
var book = {
    title: "The Principles of Object-Oriented JavaScript"
};

// is the same as

var book = Object.create(Object.prototype, {
                title: {
                    configurable: true,
                    enumerable: true,
                    value: "The Principles of Object-Oriented JavaScript",
                    writable: true
                }
            });
```

Object Inheritance: Example

```
var person1 = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};

var person2 = Object.create(person1, {
    name: {
        configurable: true,
        enumerable: true,
        value: "Greg",
        writable: true
    }
});

person1.sayName();          // outputs "Nicholas"
person2.sayName();          // outputs "Greg"
```




```
console.log(person1.hasOwnProperty("sayName"));      // true
console.log(person1.isPrototypeOf(person2));         // true
console.log(person2.hasOwnProperty("sayName"));      // false
```

ES6 features

1

Let and Const Block-Scoped Constructs Let and Const

2

Promise in ES6

3

Arrow function

Block-Scoped Constructs Let and Const

```
function calculateTotalAmount (vip) {
  var amount = 0
  if (vip) {
    var amount = 1
  }
  { // more crazy blocks!
    var amount = 100
    {
      var amount = 1000
      }
  }
  return amount
}

console.log(calculateTotalAmount(true))
```

```
function calculateTotalAmount (vip) {
  var amount = 0 // probably should also be let, but you can mix var and let
  if (vip) {
    let amount = 1 // first amount is still 0
  }
  { // more crazy blocks!
    let amount = 100 // first amount is still 0
    {
      let amount = 1000 // first amount is still 0
      }
  }
  return amount
}

console.log(calculateTotalAmount(true))
```

Promise

```
const promise = new Promise(function(resolve, reject) {
  // ... some code

  if (/* Succeed */){
    resolve(value);
  } else {
    reject(error);
  }
});
```

```
promise.then(function(value) {
  // success
}, function(error) {
  // failure
});
```

Promise

```
var wait1000 =  new Promise(function(resolve, reject) {
  setTimeout(resolve, 1000)
}).then(function() {
  console.log('Yay!')
})
```

Promise

```
let promise = new Promise(function(resolve, reject) {
  console.log('Promise');
  resolve();
});

promise.then(function() {
  console.log('resolved.');
});

console.log('Hi!');
```

Promise

```
const getJSON = function(url) {
  const promise = new Promise(function(resolve, reject){
    const handler = function() {
      if (this.readyState !== 4) {
        return;
      }
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
    const client = new XMLHttpRequest();
    client.open("GET", url);
    client.onreadystatechange = handler;
    client.responseType = "json";
    client.setRequestHeader("Accept", "application/json");
    client.send();

  });

  return promise;
};

getJSON("/posts.json").then(function(json) {
  console.log('Contents: ' + json);
}, function(error) {
  console.error('Something went wrong', error);
});
```

Arrow function

```
var wait1000 =  new Promise(function(resolve, reject) {
  setTimeout(resolve, 1000)
}).then(function() {
  console.log('Yay!')
})
```

```
var wait1000 =  new Promise((resolve, reject)=> {
  setTimeout(resolve, 1000)
}).then(()=> {
  console.log('Yay!')
})
```

Exercise

Create a constructor function Calculator that creates objects with 3 methods:

- read() asks for two values using prompt and remembers them in object properties.
- sum() returns the sum of these properties.
- mul() returns the multiplication product of these properties.

Exercise

Create a constructor function called *Hero* That will accept the arguments *name* and *occupation*.

1. Use *Hero.prototype* to add a method *whoAreYou* that will return:

   My name is [the hero's name] and I am a [the hero's occupation].
2. Use the *Hero* constructor to create an object hero1 with the *name* Michaelangello and *occupation* Ninja.
3. Use the *whoAreYou* method to log to the console hero1's *name* and *occupation*.


Any Query?