JavaScript Advanced

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

Scope

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

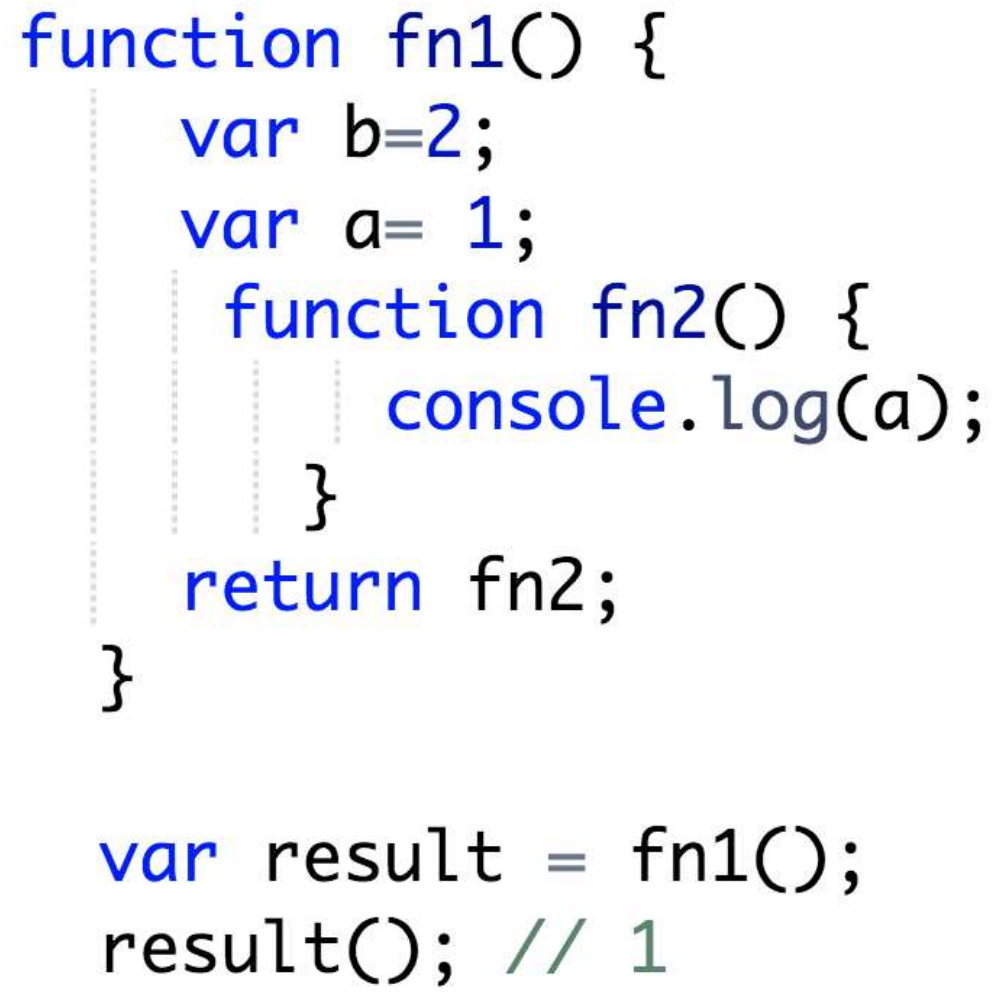
- A global variable has **global scope**: All scripts and functions on a web page can access it.

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.

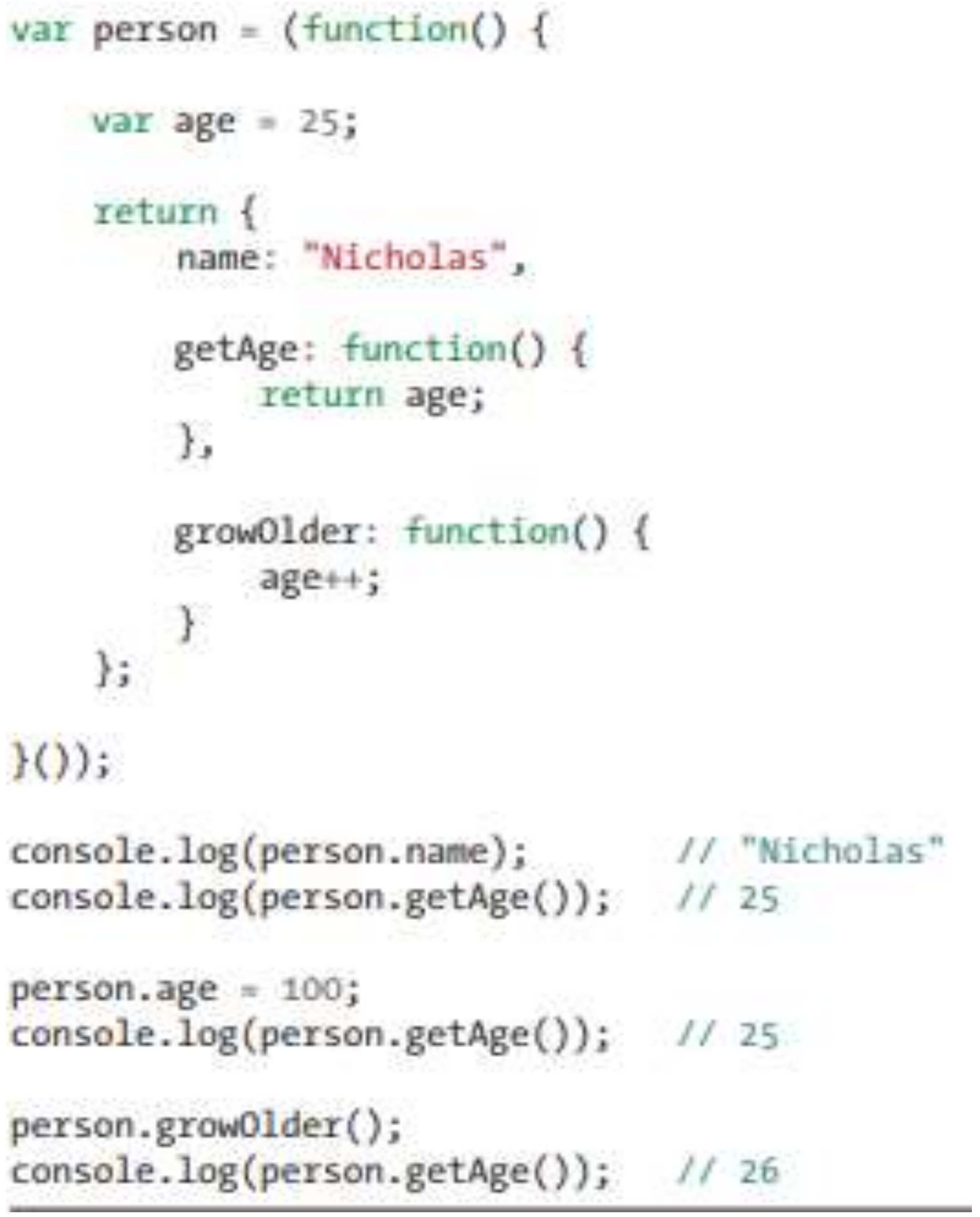
Closure

- *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



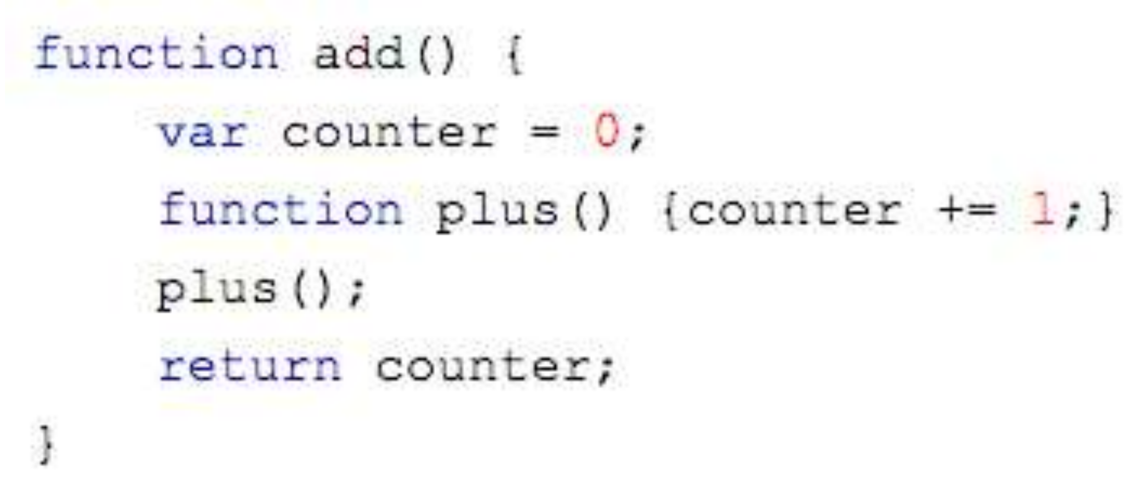
```
var person = (function() {

    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

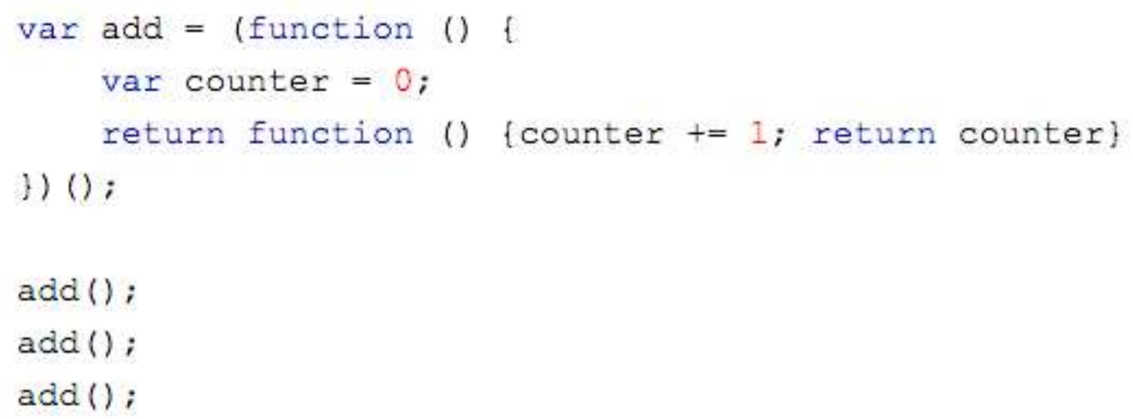
An IIFE is a function expression that is defined and then called immediately to produce a result.

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;
}
```

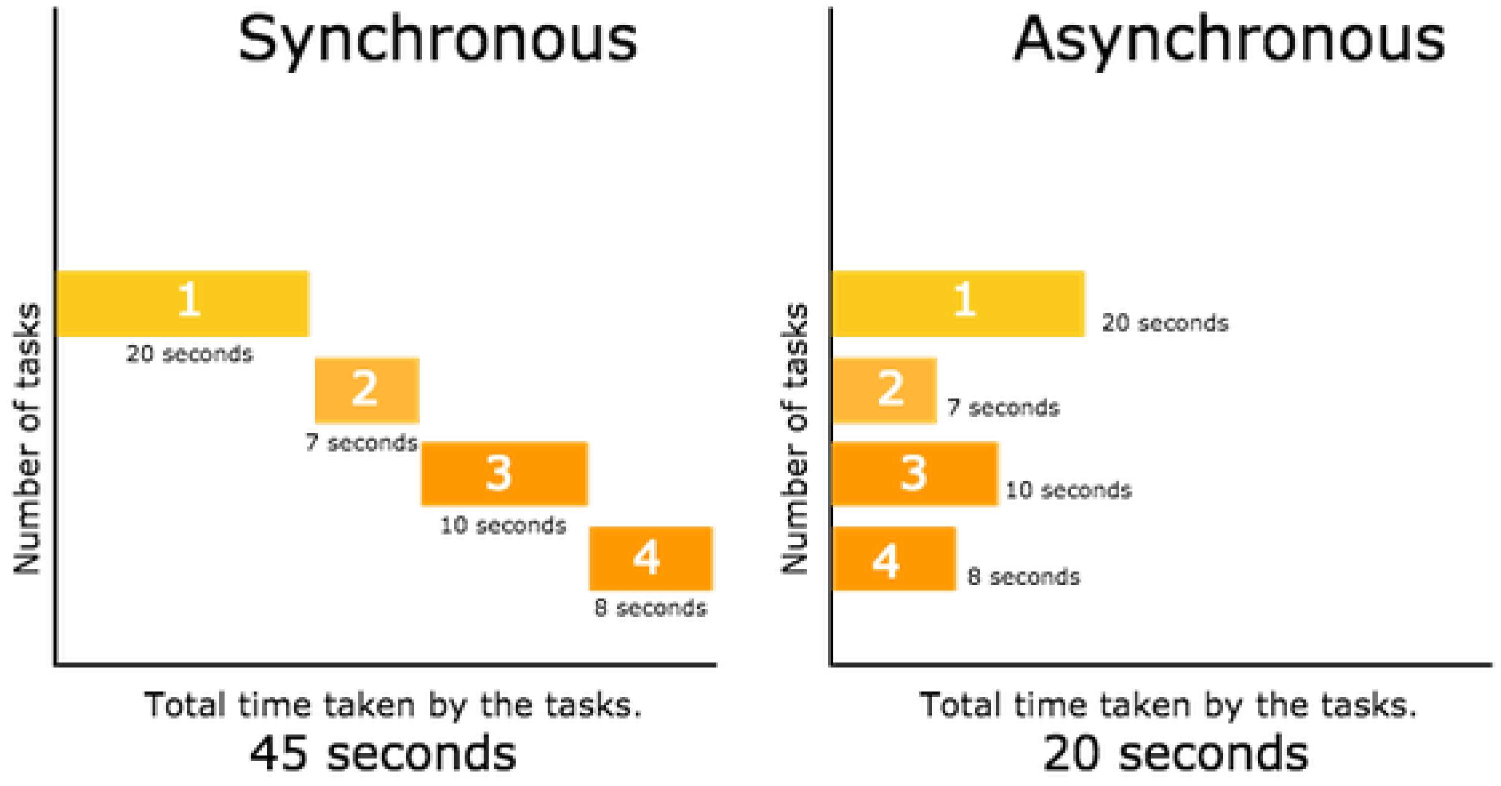


```
var add = (function () {
    var counter = 0;
    return function () {counter += 1; return counter}
})();

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

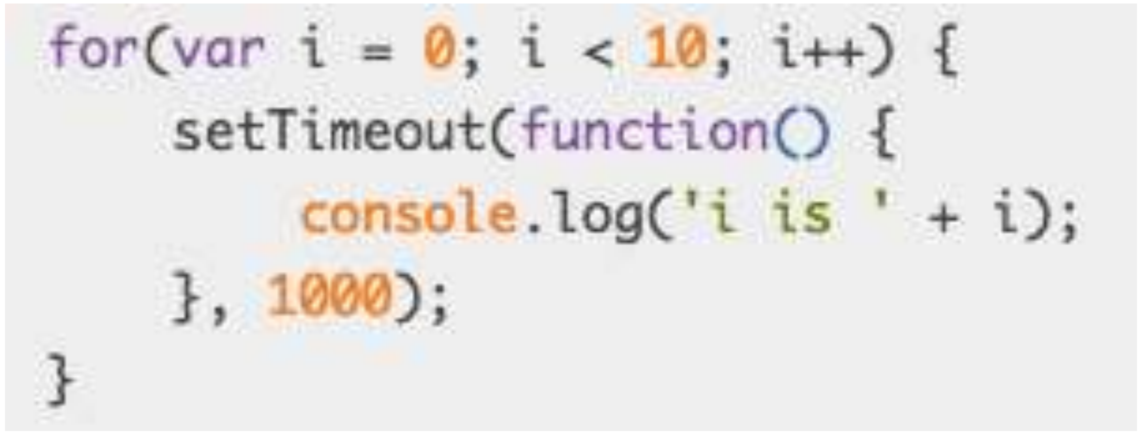
Asynchronous

- 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.

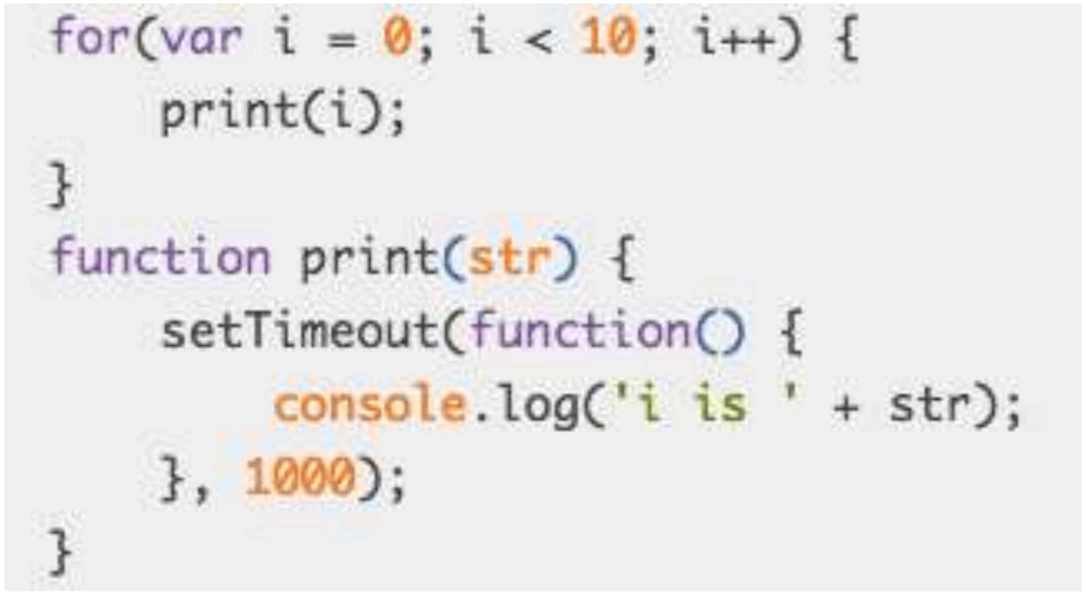
Synchronous VS Asynchronous



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.

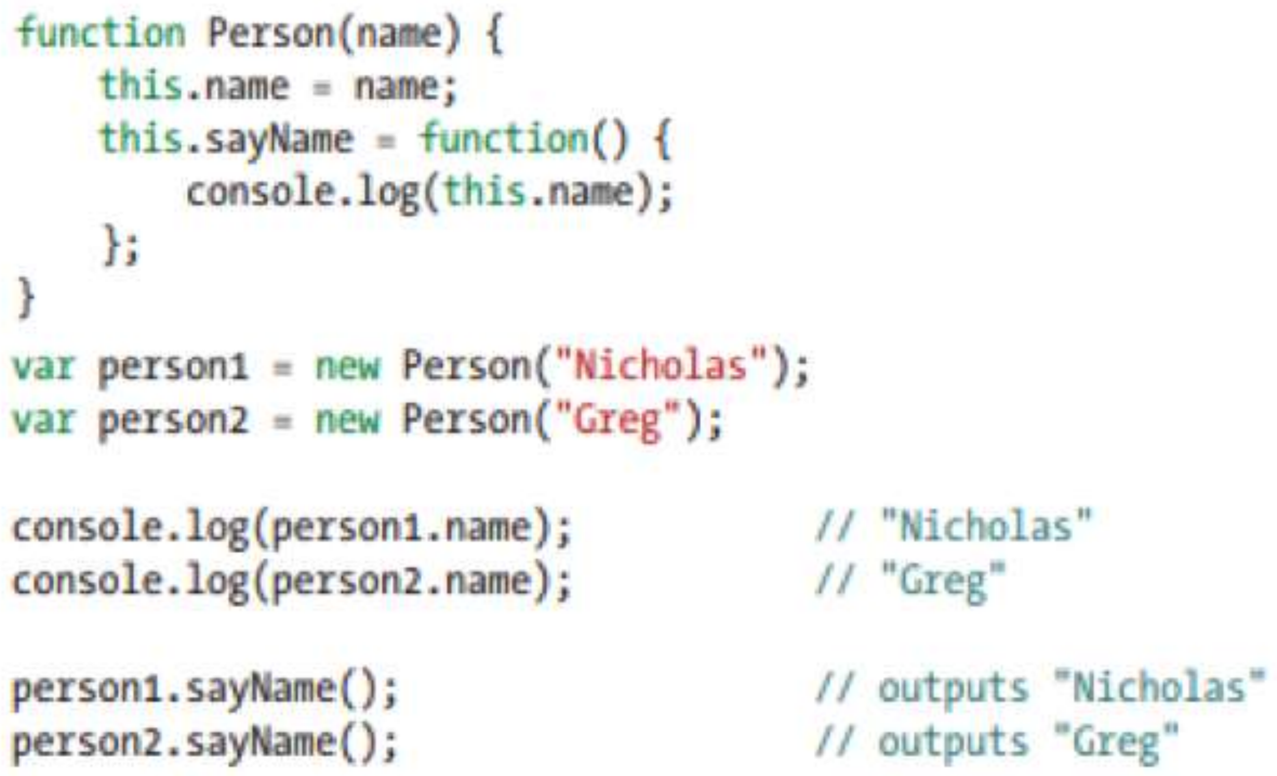
Constructor

```
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

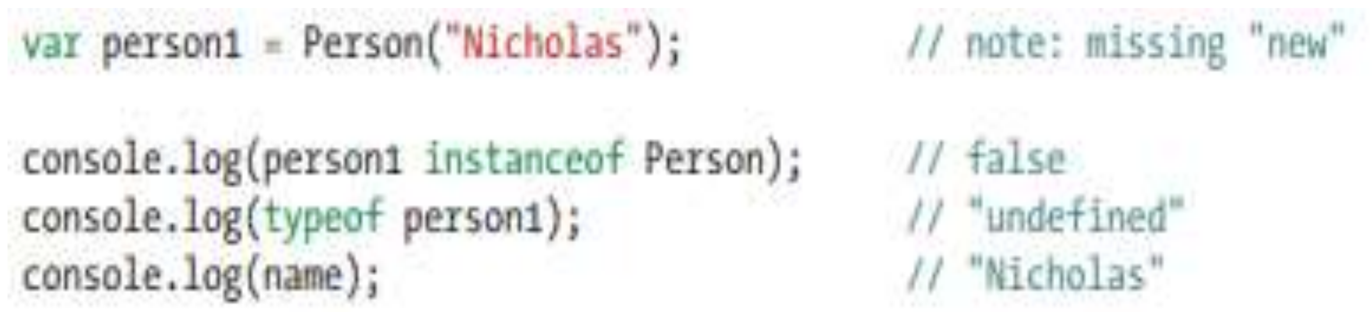
- A *constructor* with parameter, properties and methods:



```
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.

ES6 features

1

Let and Const Block-Scoped Constructs Let and Const

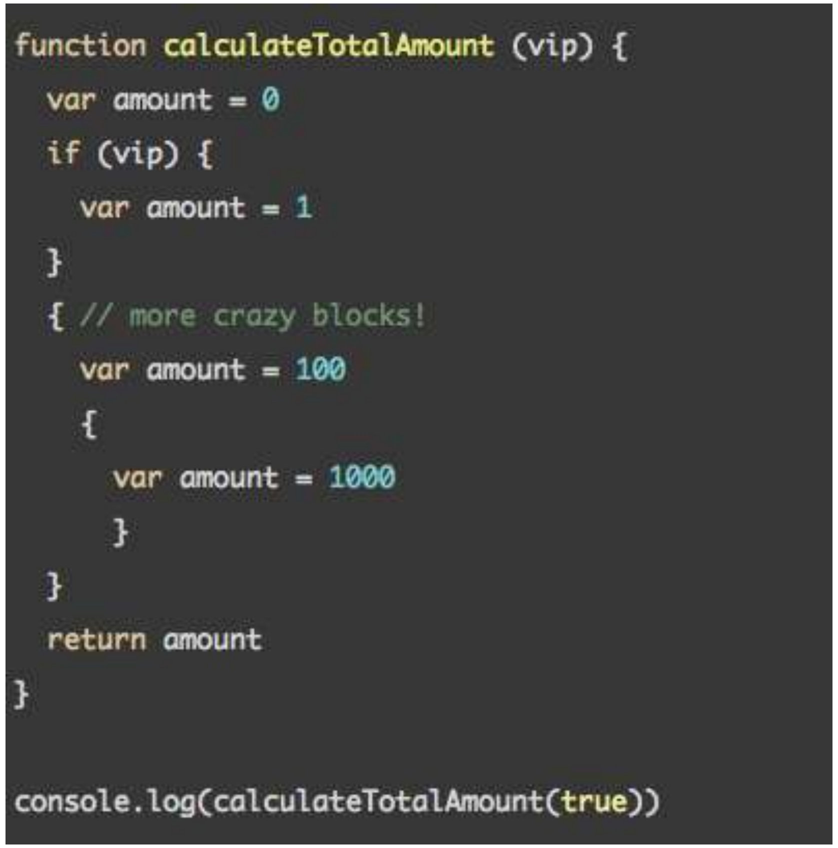
2

Promise in ES6

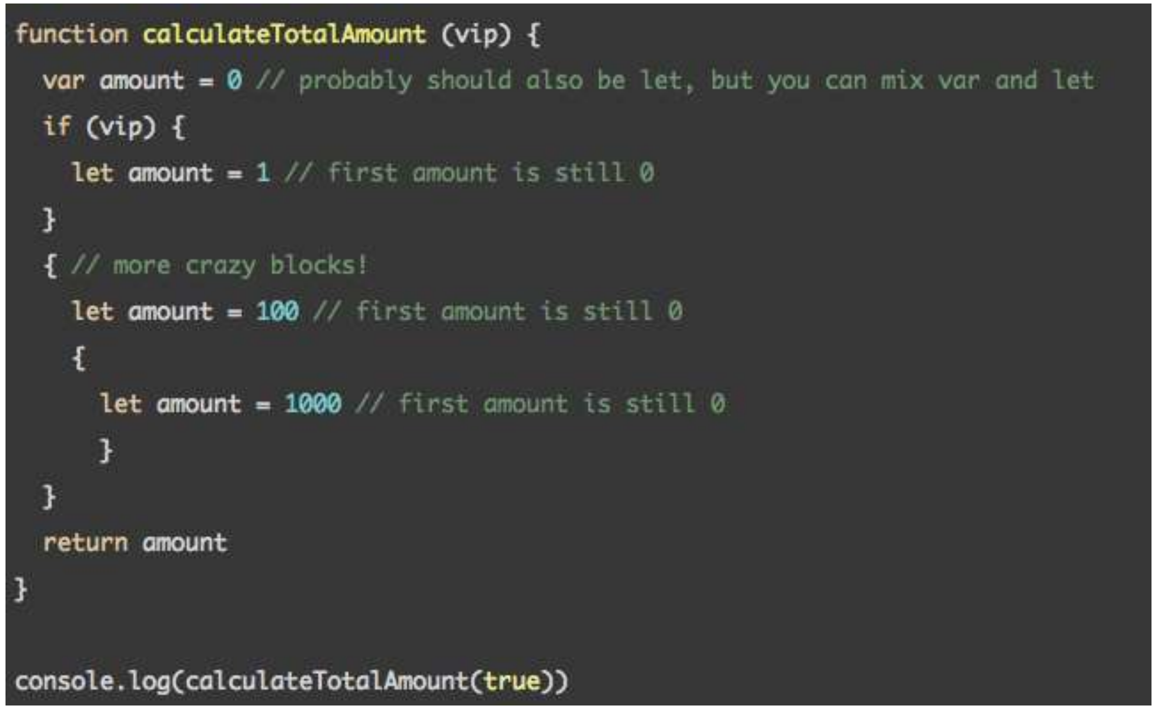
3

Arrow function

Block-Scoped Constructs Let and Const

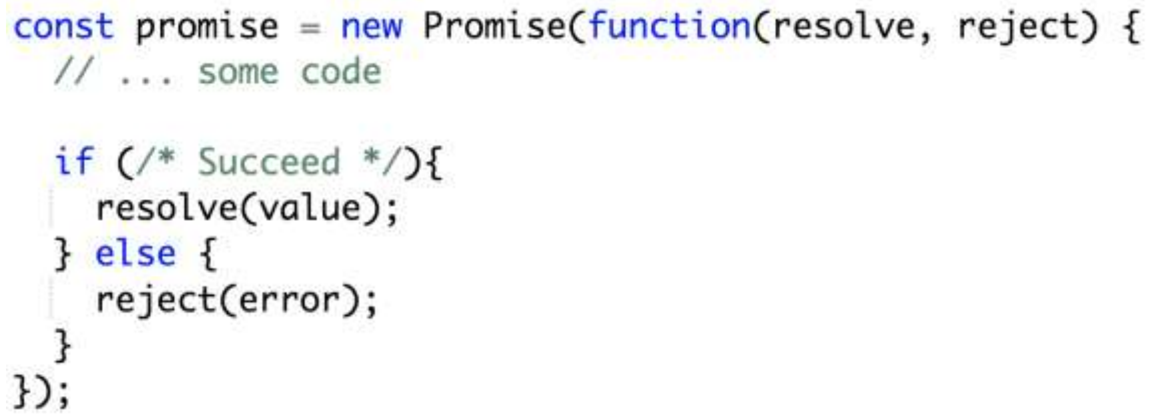


```
function calculateTotalAmount (vip) {
  var amount = 0
  if (vip) {
    var amount = 1
  }
  { // more crazy blocks!
    var amount = 100
    {
      var amount = 1000
      }
  }
  return amount
}

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

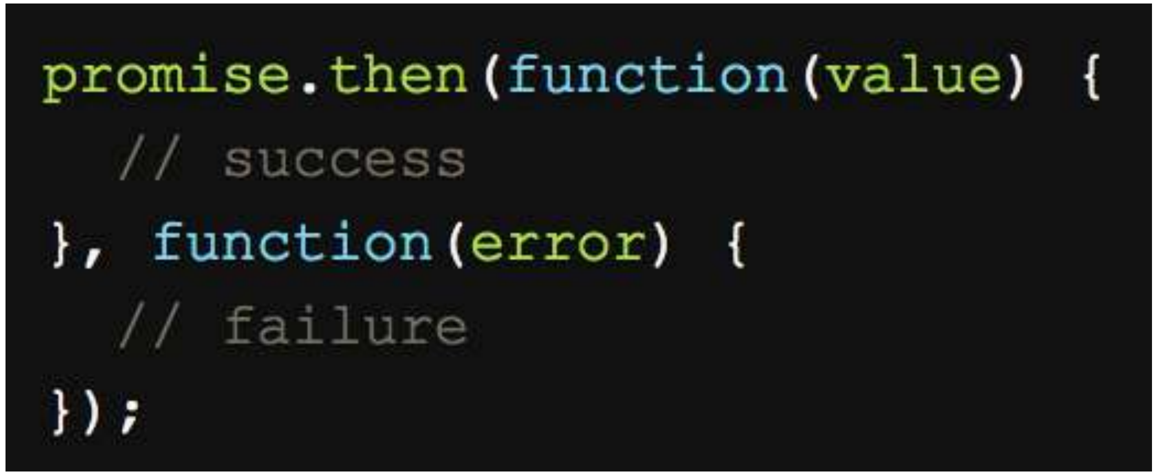


```
function calculateTotalAmount (vip) {
  var amount = 0 // probably should also be let, but you can mix var and let
  if (vip) {
    let amount = 1 // first amount is still 0
  }
  { // more crazy blocks!
    let amount = 100 // first amount is still 0
    {
      let amount = 1000 // first amount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

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

Promise

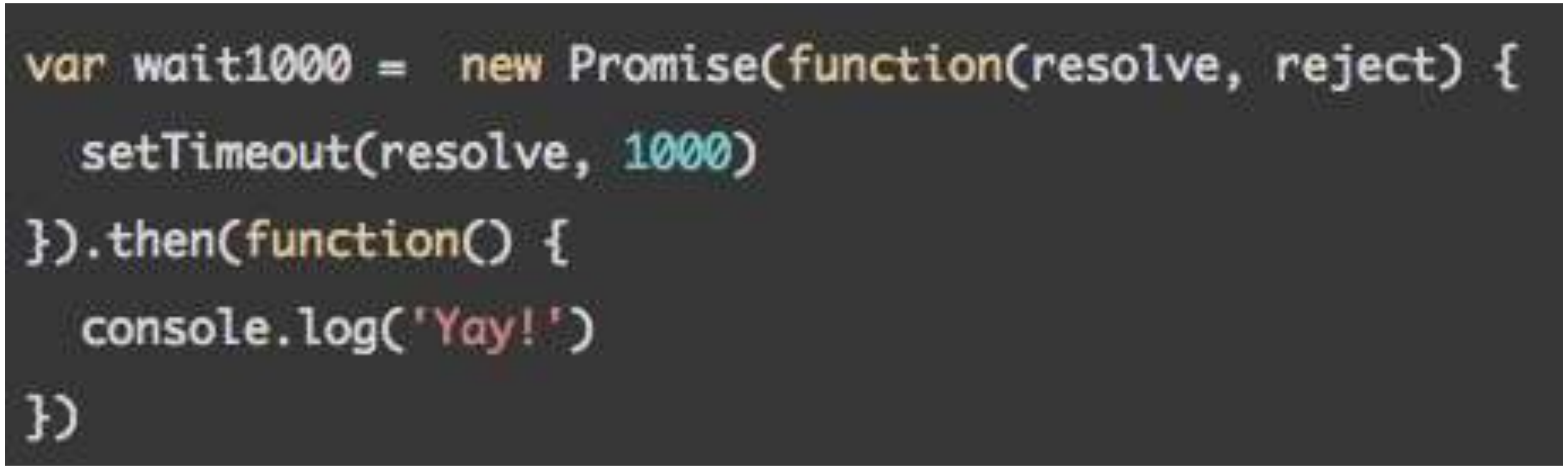


```
const promise = new Promise(function(resolve, reject) {
  // ... some code

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



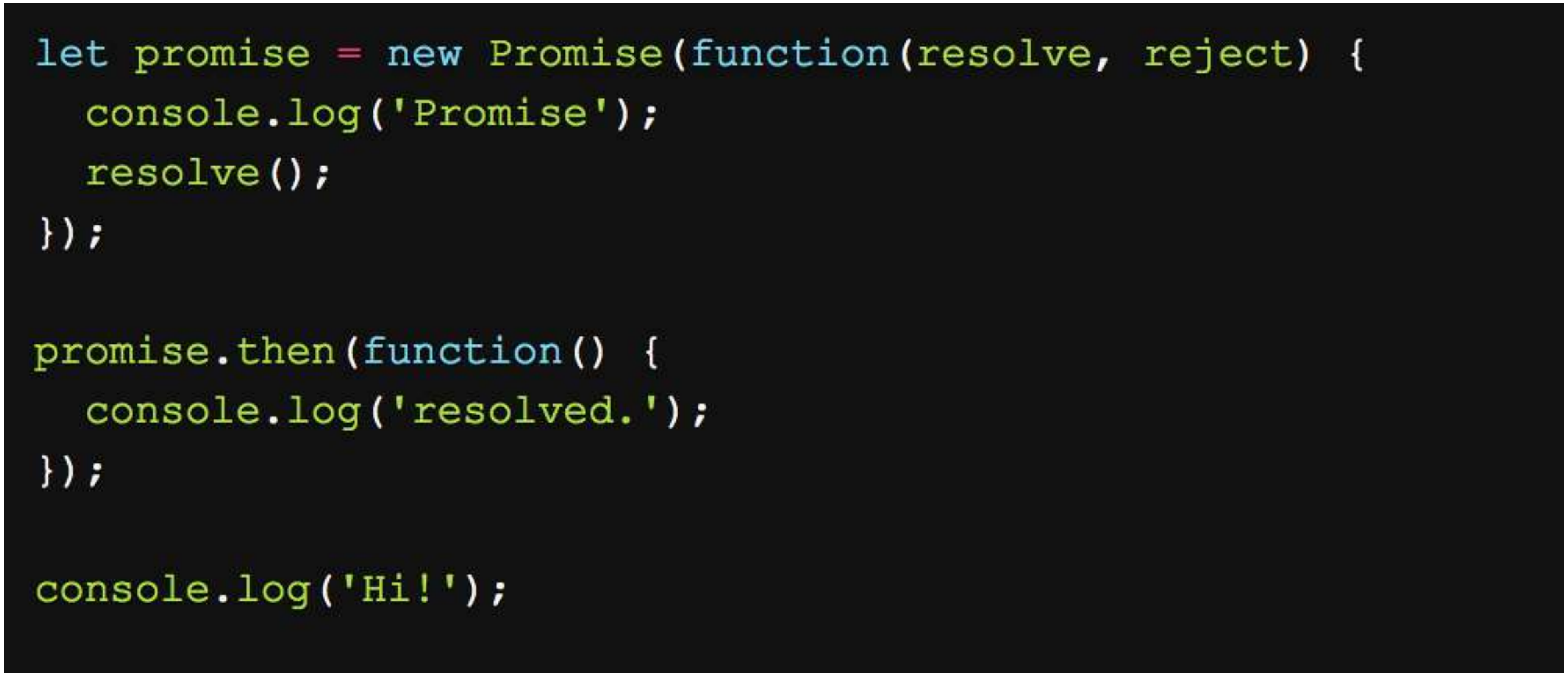
```
promise.then(function(value) {
  // success
}, function(error) {
  // failure
});
```

Promise



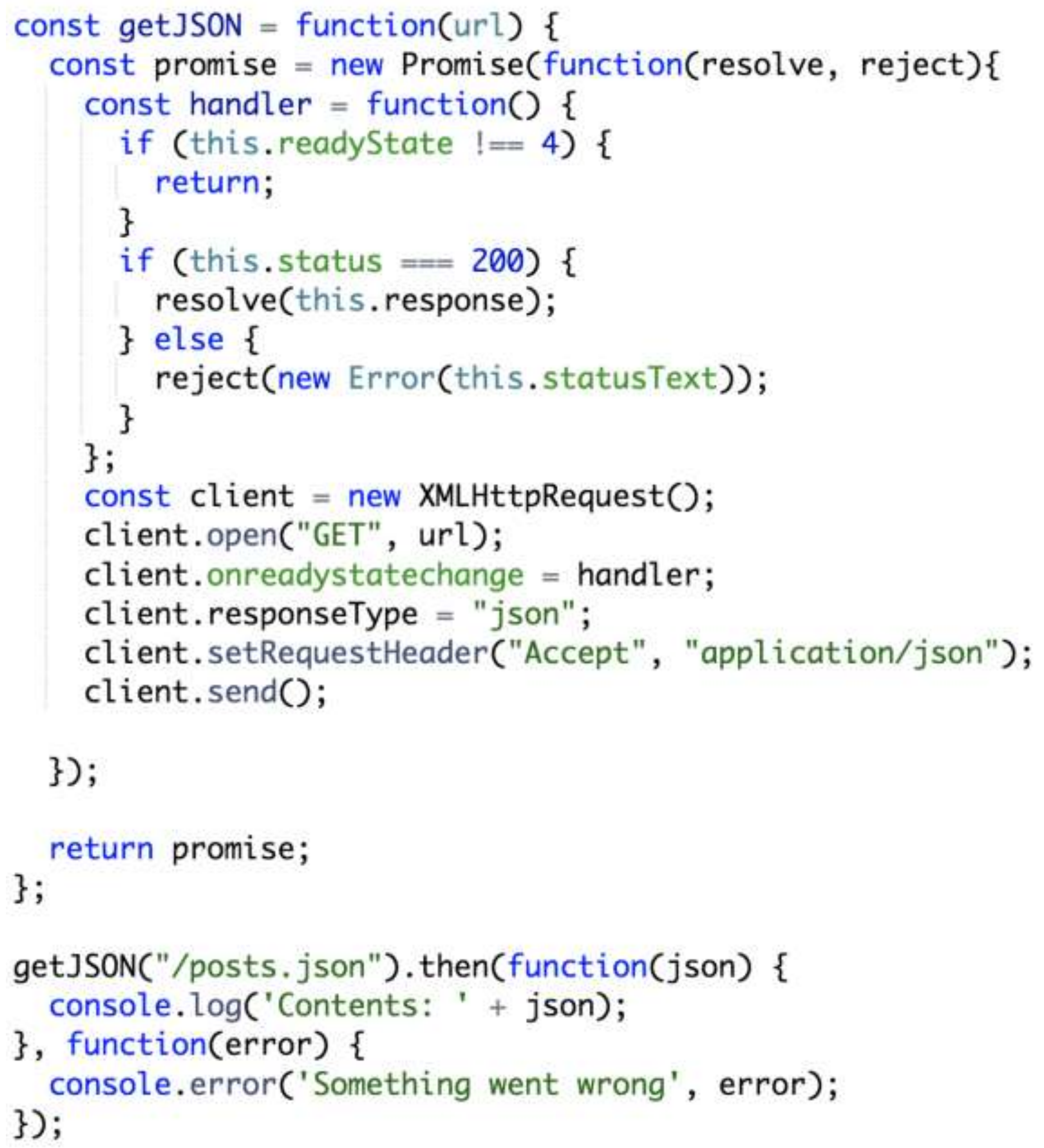
```
var wait1000 =  new Promise(function(resolve, reject) {
  setTimeout(resolve, 1000)
}).then(function() {
  console.log('Yay!')
})
```

Promise



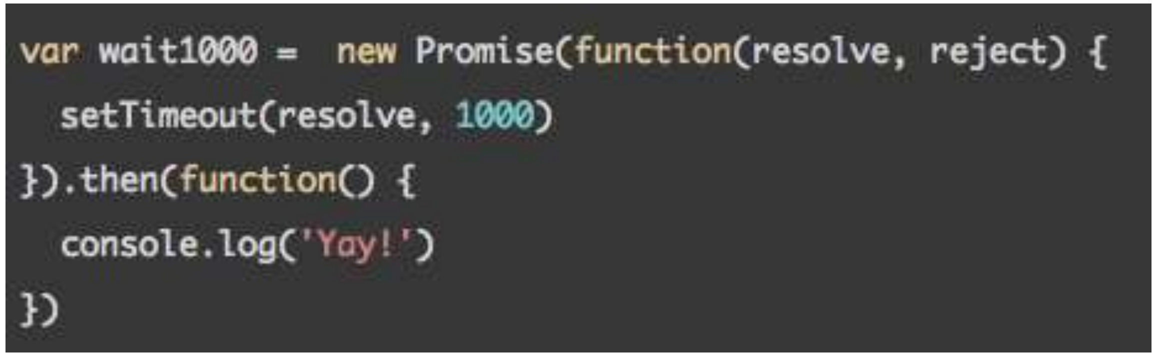
```
let promise = new Promise(function(resolve, reject) {
  console.log('Promise');
  resolve();
});

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

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

Promise

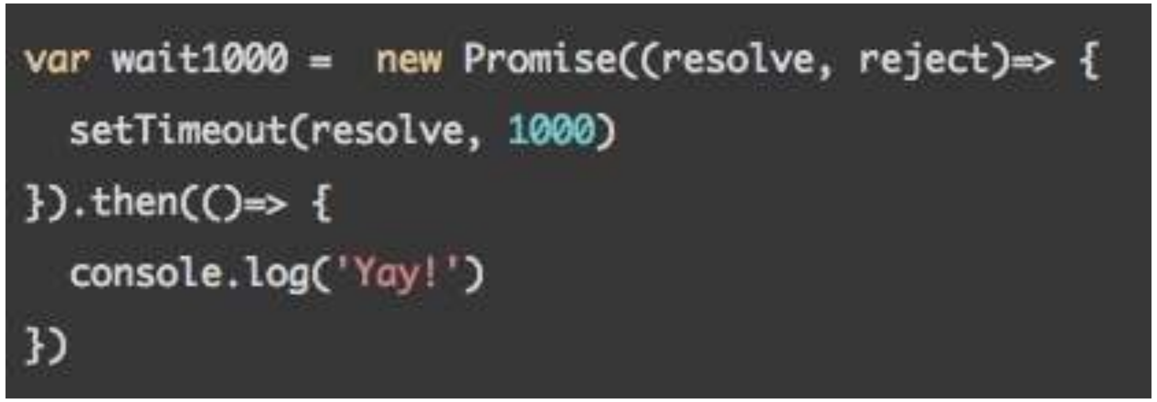


```
const getJSON = function(url) {
  const promise = new Promise(function(resolve, reject){
    const handler = function() {
      if (this.readyState !== 4) {
        return;
      }
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
    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!')
})
```