Web Application Development

2019

Javascript

Part II

Functions

Functions allow us to store a piece of code that does a single task inside a defined block, and then call that code whenever we need it using a single short command

Function declaration

```
1.  function sayHello() {
2.      alert('hello');
3.  }
```

Invoking function

```
1.   sayHello();
```

Function parameters

```
1.  function add(num1, num2) {
2.      return num1 + num2;
3.  }
4.
5.  let myNumber = add(5, 7)
6.  // 12
```

Anonymous functions

```
1.  let myFunction = function() {
2.      alert('Hello');
3.  }
4.
5.  function doSomething(string, callback) {
6.      callback();
7.      alert(string);
8.  }
9.  doSomething('World', myFunction)
10.
11. // Hello
12. // World
```

Nesting

```
1.  function addSquares(a, b) {
2.      function square(x) {
3.          return x * x;
4.      }
5.      return square(a) + square(b);
6.  }
7.
8.  let  a = addSquares(2, 3); // returns 13
9.  let  b = addSquares(3, 4); // returns 25
10. let c = addSquares(4, 5); // returns 41
```

Scopes

Scope

The current context of execution. The context in which values and expressions are "visible," or can be referenced. If a variable or other expression is not "in the current scope," then it is unavailable for use. Scopes can also be layered in a hierarchy, so that child scopes have access to parent scopes, but not vice versa.

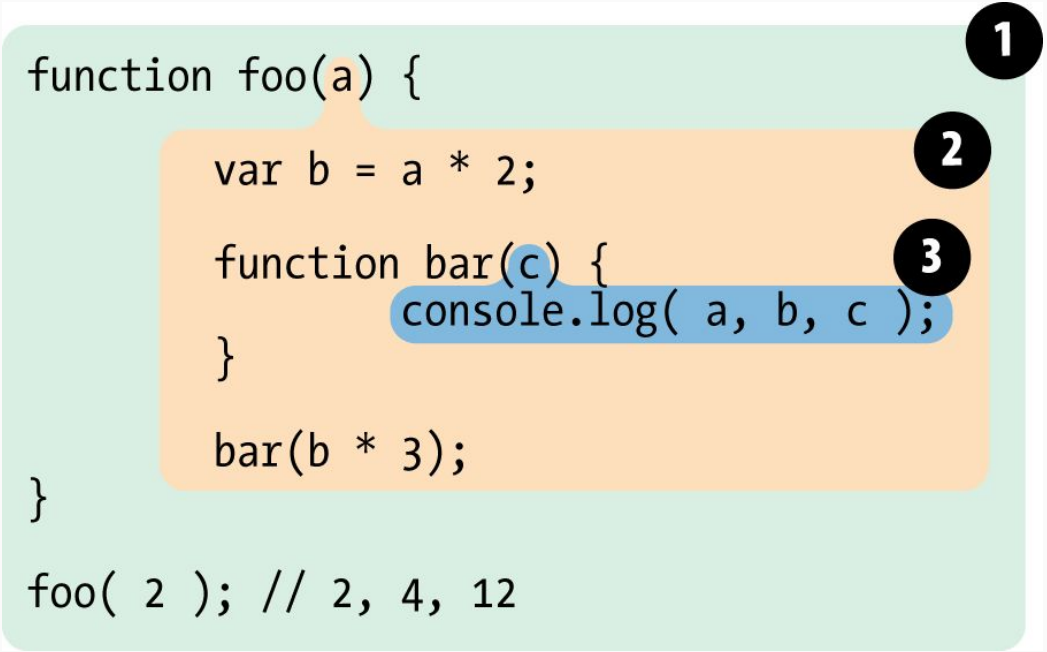
Scope

```
1.  function exampleFunction() {
2.      let x = 'declared inside function';
3.      // x can only be used in exampleFunction
4.      console.log('Inside function');
5.      console.log(x);
6.  }
7.
8.  console.log(x);  // Causes error
9.
```

Scope

```
1.  let x = 'declared outside function';
2.  function exampleFunction() {
3.      console.log('Inside function');
4.      console.log(x);
5.  }
6.
7.  console.log('Outside function');
8.  console.log(x);
```

Scope



1. The global scope, and has just one identifier in it: `foo`
2. The scope of `foo`, which includes the 3 identifiers: `a`, `bar` and `b`
3. The scope of `bar`, and it includes just one identifier: `c`

Blocks as Scopes

```
1.  var foo = true;
2.
3.  if (foo) {
4.      var bar = foo * 2;
5.      console.log( bar );
6.  }
7.
8.  for (var i=0; i<10; i++) {
9.      console.log( i );
10. }
```

var **vs** let

Blocks as Scopes

```
1.  var foo = true;
2.
3.  if (foo) {
4.      var bar = foo * 2;
5.      console.log( bar );
6.  }
7.
8.  for (var i=0; i<10; i++) {
9.      console.log( i );
10. }
11. console.log( bar ); // 2
12. console.log( i );   // 10
```

```
1.  let foo = true;
2.
3.  if (foo) {
4.      let bar = foo * 2;
5.      console.log( bar );
6.  }
7.
8.  for (let i=0; i<10; i++) {
9.      console.log( i );
10. }
11. console.log( bar ); // ERROR
12. console.log( i );   // ERROR
```

Events

Events are actions or occurrences that happen in the system you are programming — the system will fire a signal of some kind when an event occurs, and also provide a mechanism by which some kind of action can be automatically taken when the event occurs.

Events

- The user clicking the mouse over a certain element or hovering the cursor over a certain element.
- The user pressing a key on the keyboard.
- The user resizing or closing the browser window.
- A web page finishing loading.
- A form being submitted.
- A video being played, or paused, or finishing play.
- An error occurring.

Add an event

```
1.  <button>Change color</button>
```

```
1.  var button = document.querySelector('button');
2.  function backgroundChange() {
3.    let randomColor= 'rgb('
4.        + random(255) + ','
5.        + random(255) + ','
6.        + random(255) + ')';
7.    document.body.style.backgroundColor = randomColor;
8.  }
9.  button.onclick = backgroundChange;
```

Add an event

```
<button onclick="backgroundChange()">
    Change color
</button>
```

```
var button = document.querySelector('button');
function backgroundChange() {
  let randomColor= 'rgb('
      + random(255) + ','
      + random(255) + ','
      + random(255) + ')';
  document.body.style.backgroundColor = randomColor;
}
```

Add an event

```
1.  <button>Change color</button>
```

```
1.  var button = document.querySelector('button');
2.  function backgroundChange() {
3.    let randomColor= 'rgb('
4.        + random(255) + ','
5.        + random(255) + ','
6.        + random(255) + ')';
7.    document.body.style.backgroundColor = randomColor;
8.  }
9.  button.addEventListener('click', backgroundChange);
```

Remove an event

```
1.  button.removeEventListener('click', backgroundChange);
```

Preventing default behavior

Sometimes, you'll come across a situation where you want to stop an event doing what it does by default.

```
1.  var form = document.querySelector('form');
2.  var name = document.getElementById('name');
3.
4.  form.onsubmit = function(e) {
5.      if (name.value === '') {
6.          e.preventDefault();
7.      }
8.  }
```

Bubbling and capturing

When an event is fired on an element that has parent elements, browsers run two different phases — the **capturing** phase and the **bubbling** phase.

In the **capturing** phase:

- The browser checks to see if the element's outermost ancestor (`<html>`) has an `onclick` event handler registered on it in the capturing phase, and runs it if so.
- Then it moves on to the next element inside `<html>` and does the same thing, then the next one, and so on until it reaches the element that was actually clicked on.

In the **bubbling** phase, the exact opposite occurs:

- The browser checks to see if the element that was actually clicked on has an `onclick` event handler registered on it in the bubbling phase, and runs it if so.
- Then it moves on to the next immediate ancestor element and does the same thing, then the next one, and so on until it reaches the `<html>` element.

Fixing the problem with `stopPropagation()`

This is annoying behavior, but there is a way to fix it! The standard event object has a function available on it called `stopPropagation()`, which when invoked on a handler event object makes it so that handler is run, but the event doesn't bubble any further up the chain, so no more handlers will be run.

```
1.  video.onclick = function(e) {
2.      e.stopPropagation();
3.      video.play();
4.  }
```

Bubbling and capturing

Note: Why bother with both capturing and bubbling? Well, in the bad old days when browsers were much less cross-compatible than they are now, Netscape only used event capturing, and Internet Explorer used only event bubbling. When the W3C decided to try to standardize the behavior and reach a consensus, they ended up with this system that included both, which is the one modern browsers implemented.

Note: All event handlers are registered in the bubbling phase, and this makes more sense most of the time. If you really want to register an event in the capturing phase instead, you can do so by registering your handler using `addEventListener()`, and setting the optional third property to `true`.

Objects

Object

```
1.  const person = {
2.      name: ['Bob', 'Smith'],
3.      age: 32,
4.      gender: 'male',
5.      interests: ['music', 'skiing'],
6.      bio: function() {
7.          alert(this.name[0] + ' ' + this.name[1]);
8.      }
9.  };
```

Dot notation

```
1.  person.name
2.  person.name[0]
3.  person.age
4.  person.bio()
```

Bracket notation

```
1.  person['name'][0]
2.  person['name'][1]
3.  person['age']
```

Setting object members

```
person.age = 45;
person['name'][1] = 'Doe';
person['eyes'] = 'hazel';
person.farewell = function() {
    alert("Bye everybody!");
}
```

“`This`” is important

The `this` keyword refers to the current object the code is being written inside

```
1.  const person1 = {
2.      name: 'Chris',
3.      greeting: function() {
4.          alert('Hi! I`m ' + this.name + '.');
5.      }
6.  }
7.  const person2 = {
8.      name: 'Brian',
9.      greeting: function() {
10.         alert('Hi! I`m ' + this.name + '.');
11.     }
12. }
```

Object constructor

```
1.  function Person(first, last, age, gender) {
2.      this.name = {
3.          first : first,
4.          last : last
5.      };
6.      this.age = age;
7.      this.gender = gender;
8.      this.bio = function() {
9.          alert(this.name.first + ' '
10.             + this.name.last
11.             + ' is ' + this.age + ' years old.');
12.     };
13. this.greeting = function() {
14.     alert('Hi! I`m ' + this.name.first + '.');
15.     };
16. }
17. const bob = new Person('Bob', 'Smith', 32, 'male')
```

Promises

Promises

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

Old-style :

```
1.  function success(result) {
2.      console.log("Audio file ready: " + result);
3.  }
4.  function fail(error) {
5.      console.error("Error generating file:" + error);
6.  }
7.  createAudioFileAsync(settings, success, fail);
```

Promises

Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

Modern style...

```
1.  const promise = createAudioFileAsync(settings);
2.  promise.then(success).catch(fail);
```

Guarantees

Unlike "old-style", passed-in callbacks, a promise comes with some guarantees:

- Callbacks will never be called before the completion of the current run of the JavaScript event loop.
- Callbacks added with `then`() even *after* the success or failure of the asynchronous operation, will be called, as above.
- Multiple callbacks may be added by calling `then`() several times. Each callback is executed one after another, in the order in which they were inserted.

Chaining

A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step. We accomplish this by creating a **promise chain**.

```
doSomething()
    .then(function(result) {
        return doSomethingElse(result);
    })
    .then(function(newResult) {
        return doThirdThing(newResult);
    })
    .then(function(finalResult) {
        console.log('Final result: ' + finalResult);
    })
    .catch(failureCallback);
```

Chaining

old-style

A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step. We accomplish this by creating a **promise chain**.

```
doSomething(function(a) {
    doSomethingElse(a, function(b) {
        doThirdThing(b, function(c) {
            console.log('Final: ' + c);
        }, failureCallback);
    }, failureCallback);
}, failureCallback);
```

Composition

`Promise.resolve()` and `Promise.reject()` are shortcuts to manually create an already resolved or rejected promise respectively. This can be useful at times.

`Promise.all()` and `Promise.race()` are two composition tools for running asynchronous operations in parallel.

```
1.  Promise.all([func1(), func2(), func3()])
2.      .then(function([result1, result2, result3]){
3.              /* use result1, result2 and result3 */
4.          }
5.      );
```

Iterators

Iterators

In JavaScript an **iterator** is an object which defines a sequence and potentially a return value upon its termination. More specifically an iterator is any object which implements the Iterator protocol by having a `next()` method which returns an object with two properties: `value`, the next value in the sequence; and `done`, which is `true` if the last value in the sequence has already been consumed. If `value` is present alongside `done`, it is the iterator return value.

```
1.  function rangeIterator(start=0, end=100, step=1){
2.      let next = start;
3.      let count = 0;
4.      const iterator = {
5.          next: function() {
6.              let result;
7.              if (next < end) {
8.                  result = {value: next, done: false}
9.                  next += step;
10.                 count++;
11.                 return result;
12.             }
13.             return { value: count, done: true }
14.         }
15.     }; return iterator;
16. }
```

Iterators

In JavaScript an **iterator** is an object which defines a sequence and potentially a return value upon its termination. More specifically an iterator is any object which implements the Iterator protocol by having a `next()` method which returns an object with two properties: `value`, the next value in the sequence; and `done`, which is `true` if the last value in the sequence has already been consumed. If `value` is present alongside `done`, it is the iterator return value.

```
1.  let it = rangeIterator(1, 10, 2);
2.  let result = it.next();
3.  while (!result.done) {
4.      console.log(result.value); // 1 3 5 7 9
5.      result = it.next();
6.  }
```

Generators

Generator functions

While custom iterators are a useful tool, their creation requires careful programming due to the need to explicitly maintain their internal state. Generator functions provide a powerful alternative: they allow you to define an iterative algorithm by writing a single function whose execution is not continuous.

Generator functions are written using the function* syntax. When called initially, generator functions do not execute any of their code, instead returning a type of iterator called a Generator. When a value is consumed by calling the generator's `next` method, the Generator function executes until it encounters the yield keyword.

```
1.  function* rangeIterator(start=0, end=100, step=1) {
2.      for (let i = start; i < end; i += step) {
3.          yield i;
4.      }
5.  }
```

Generators

```
1.  function* fibonacci() {
2.      var fn1 = 0;
3.      var fn2 = 1;
4.      while (true) {
5.          var current = fn1;
6.          fn1 = fn2;
7.          fn2 = current + fn1;
8.          var reset = yield current;
9.          if (reset) {
10.             fn1 = 0;
11.             fn2 = 1;
12.         }
13.     }
14. }
```

Generators

```
1.  var sequence = fibonacci();
2.
3.  console.log(sequence.next().value); // 0
4.  console.log(sequence.next().value); // 1
5.  console.log(sequence.next().value); // 1
6.  console.log(sequence.next().value); // 2
7.  console.log(sequence.next().value); // 3
8.  console.log(sequence.next().value); // 5
9.  console.log(sequence.next().value); // 8
10. console.log(sequence.next(true).value); // 0
11. console.log(sequence.next().value); // 1
12. console.log(sequence.next().value); // 1
13. console.log(sequence.next().value); // 2
```

References

https://developer.mozilla.org/en-US/

Questions?

Next: Javascript Part II