Web Application Programming

Javascript Module Pattern

These slides were drawn from the following references:

http://toddmotto.com/everything-you-wanted-to-know-about-javascript-scope/

http://toddmotto.com/mastering-the-module-pattern/

http://www.w3schools.com/js/js_function_closures.asp

Global Scope

- Before you write a line of JavaScript, you are in the *Global Scope*. If we declare a variable, it is defined globally:

```
// global scope
var name = 'Todd';
```

- You will run into no issues with global scope problems (usually namespace clashes) if you control your scopes properly.
- Global Scope is needed to access functions defined in other files (such as jQuery functions).

Local Scope

- There is one global scope, and each function defined has its own (nested) local scope.
- If we define a function and create variables inside it, those variables are locally scoped:

```
// Scope A: Global scope out here var
myFunction = function () {
    // Scope B: Local scope in here
};
```

- Any locally scoped items are not visible in the global scope.

Scope Error

```
var myFunction = function () {
  var name = 'Todd'; console.log(name);
  // Todd
  };
// Error: name is not defined
console.log(name);
```

The variable *name* is scoped locally, it is not exposed to the parent scope and therefore undefined.

Function Scope

- All scopes in JavaScript are created with Function Scope *only.*
- Scopes are not created by *for* or *while* loops or expression statements like *if* or *switch*.

```
// Scope A
var myFunction = function () {
    // Scope B
    var myOtherFunction = function () {
        // Scope C
    };
};
```

Lexical Scope

When a function is within another function, the inner function has access to the scope in the outer function:

```
// Scope A
var myFunction = function () {
  // Scope B
  var name = 'Todd'; // defined in Scope B
  var myOtherFunction = function () {
     // Scope C: “name” accessible here!
  };
};
```

Global Variables

- A function can access all variables defined **inside** the function:

```
function myFunction() {
    var a = 4;
    return a * a;
}
```

- A function can also access variables defined **outside** the function**:**

```
var a = 4;
function myFunction() {
    return a * a;
}
```

Global and Local Variables

- In a web page, global variables belong to the window object.
- Global variables can be used (and changed) by all scripts in the page (and in the window).
- A local variable can only be used inside the function where it is defined. It is hidden from other functions and other scripting code.
- Global and local variables with the same name are different variables. Modifying one, does not modify the other.

Variable Lifetime

- *Global* variables live as long as your application (your window / your web page) lives.
- *Local* variables have short lives. They are created when the function is invoked, and deleted when the function is finished.

Hoisting

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current function. The following both give the same result:

Example 1

```
x = 5;
elem = document.getElementById("demo");
elem.innerHTML = x;
var x;    // Declare x
```

Example 2

```
var x;    // Declare x
x = 5;
elem = document.getElementById("demo");
elem.innerHTML = x;
```

Hoisting

JavaScript only hoists declarations, not initializations. The following Examples are equivalent:

Example 1

```
var x = 5;   // Initialize x
elem = document.getElementById("demo");
elem.innerHTML = x + " " + y;
var y = 7;   // Initialize y
```

Example 2

```
var x = 5;   // Initialize x
var y;       // Declare y
elem = document.getElementById("demo");
elem.innerHTML = x + " " + y;
y = 7;       // Assign 7 to y
```

Counter Dilemma

- Suppose you want to use a variable for counting something, and you want this *counter* to be available to all functions.
- You could use a global variable, and a function to increase the *counter*:

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

- The problem is, that any script on the page can change the counter, without calling *add* ().

Counter Inside a Function

- If the *counter* is declared inside the function, nobody will be able to change it without calling *add*():

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

- It does not work! Every time the *add*() function is called, the *counter* is set to 1.

Closure Solution

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

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

A *closure* is a function having access to the parent scope, even after the parent function has closed.

Closure Counter

- The variable **add** is assigned the return value of a self-invoking function.
- The self-invoking function only runs once. It sets the *counter* to zero (0), and returns a function expression.
- This way **add** becomes a function. It can access the counter in the parent scope.
- This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.
- The *counter* is protected by the scope of the anonymous function, and can only be changed using the **add** function.

Closure Example

```
var sayHello = function (name) {
  var text = 'Hello, ' + name;
  return function () {
     console.log(text);
  };
};
```

The function *sayHello* returns a function, which means it needs assignment, and *then* calling:

```
var helloTodd = sayHello('Todd');
helloTodd();
```

Modules

- Immediately-Invoked-Function-Expression:

```
(function () {
   // code
})();
```

- Module:

```
var Module = (function () {
                  // code
              })();
```

Private Methods

```
var Module = (function () {

  let privateMethod = function () {
     // do something
  };

})();
```

privateMethod is locally declared inside the new scope.

Returning an Object

```
var Module = (function () {

  return {
     publicMethod: function () {
                // code
     }
  };

})();
```

To call the function: `Module.publicMethod();`

Anonymous Object Literal Return

```
var Module = (function () {
let privateMethod = function () {};
  return {
     publicMethodOne: function () {
       // I can call `privateMethod()`
     },
     publicMethodtwo: function () { },
     publicMethodThree: function () {}
  };
})();
```

Locally Scoped Object Literal

```
var Module = (function () {
  // locally scoped Object
  let myObject = {}; // "private" let

  privateMethod = function () {};

  myObject.someMethod = function () {
     // Public Method
  };

  return myObject;
```

Stacked locally scoped Object Literal

```
var Module = (function () {
  let privateMethod = function () {};

  let myObject = {
     someMethod: function () { },
     anotherMethod: function () { }
  };

  return myObject;

})();
```

Revealing Module Pattern

```
var Module = (function () {
  let privateMethod = function () {
      // private
  };
  let someMethod = function () {
      // public
  };
  let anotherMethod = function () {
      // public
  };
  return {
      someMethod: someMethod,
      anotherMethod:  anotherMethod
  };
  })();
```

Accessing Private Methods

```
var Module = (function () {
   let privateMethod = function (message) {
       console.log(message);
   };
   let publicMethod = function (text) {
       privateMethod(text);
   };
   return { publicMethod: publicMethod };
})();

// Example of passing data into a private method
// Private method will console.log() 'Hello!'
Module.publicMethod('Hello!');
```

Access Private Variables

```
var Module = (function () {
  let privateArray = [];

  let publicMethod = function (something) {
     privateArray.push(something);
  };

  return { publicMethod: publicMethod };

})();
```

Extending Modules

```
var Module = (function () {
   let privateMethod = function () {
       // private
   };
   let someMethod = function () {
       // public
   };
   let anotherMethod = function () {
       // public
   };
   return { someMethod: someMethod,
              anotherMethod: anotherMethod  };
})();
Module.extension = function () {
       // another method!
   };
```

Javascript Promises

The Promise Object

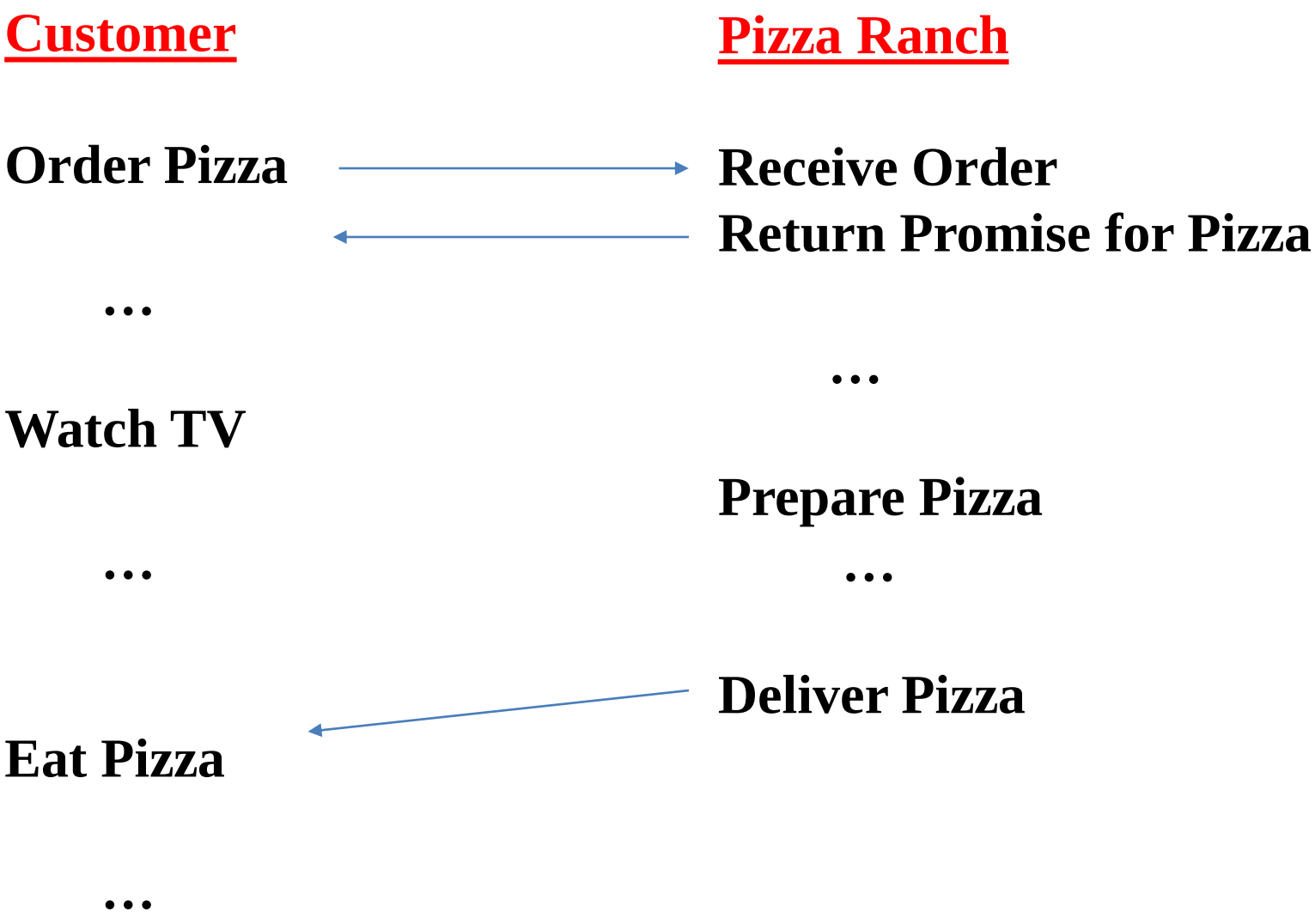
The **`Promise`** object is used for asynchronous computations. A **`Promise`** represents a value which may be available now, or in the future, or never.

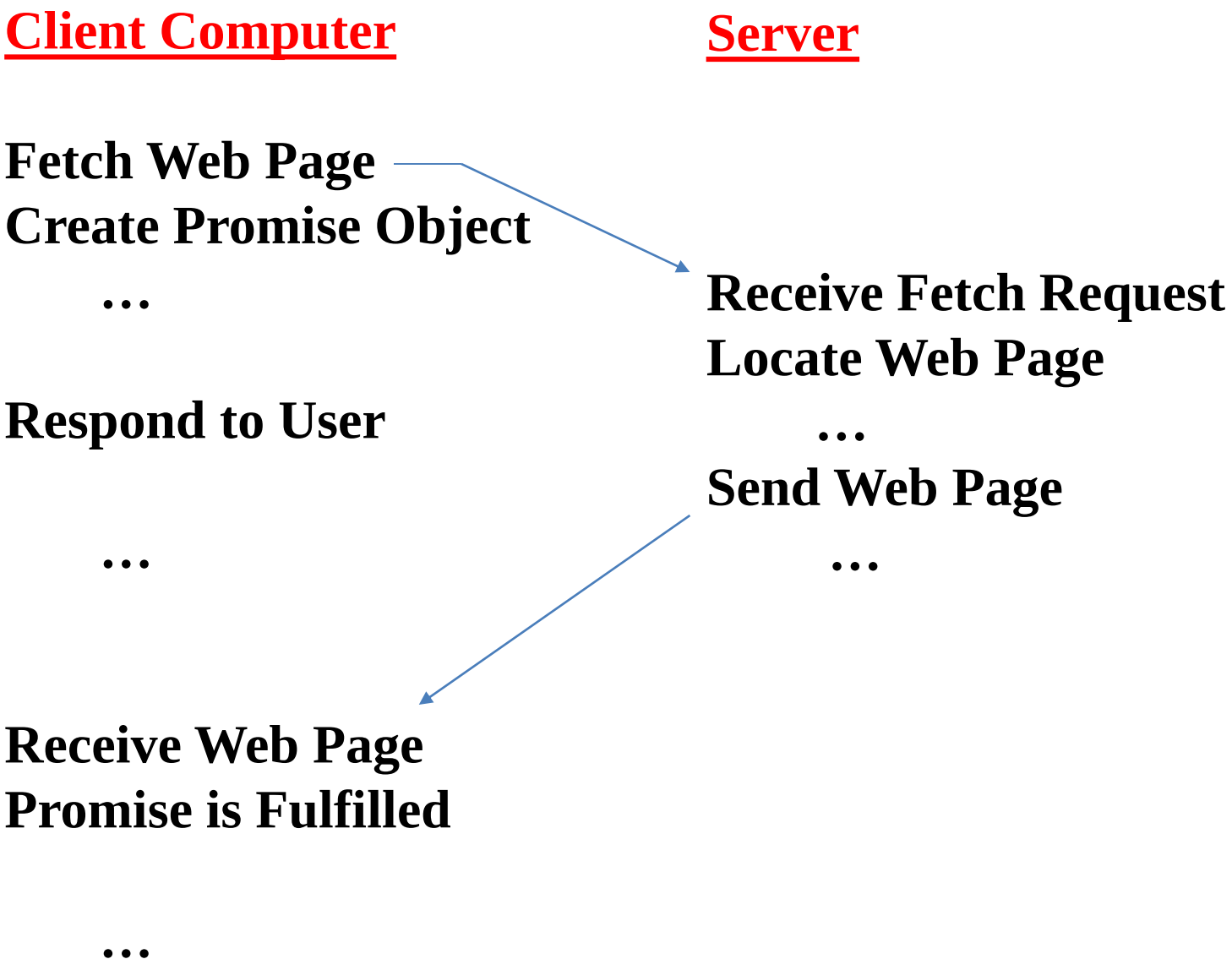
```
new Promise( function(resolve, reject) {... resolve() ... reject() } );
```

A Promise has one of these states:

- **pending**: initial state, not fulfilled or rejected.
- **fulfilled**: meaning that the operation completed successfully.
- **rejected**: meaning that the operation failed.

A pending promise can either be fulfilled with a value, or rejected with a reason (error).


Customer
Pizza Ranch
Order Pizza
Receive Order
Return Promise for Pizza
…
…
Watch TV
Prepare Pizza
…
…
Deliver Pizza
Eat Pizza
…


Client Computer
Server
Fetch Web Page
Create Promise Object
…
Respond to User
…
Receive Fetch Request
Locate Web Page
…
Send Web Page
…
Receive Web Page
Promise is Fulfilled
…

Description

- A `Promise` is a proxy for a value not necessarily known when the promise is created.
  - It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.
  - This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to *supply* the value at some point in future.

How Promises can make our code easy to read

The **`Promise`** object has two methods, **`then`** and **`catch`**. The methods will later be called depending on the state (fulfilled or rejected) of the Promise Object.

```
const postsPromise = fetch('http://mywebsite.com/API'); // return Promise

postsPromise.then(data => { console.log(data) }) // After data received
            .catch(err => { console.error(err); }) // in case rejected
```

Fetch request with Chaining

```
fetch('http://example.com/movies.json')
    .then(function(response) {
        return response.json();
         // response.json() extracts json from response
         //   and returns new Promise
    })
    .then(function(myJson) {
        console.log(JSON.stringify(myJson));
    });
```

Promise

- A Promise is an object representing the *eventual* completion or failure of an asynchronous operation.
- Most of the time in our application, we consume promises returned from calling some other APIs like `fetch()` or `json()`
  - But, you can easily create and return Promise from your own APIs.

Creating a promise

```
var giveMePizza = function(){
     return new Promise(function(resolve,reject){
          if(everythingWorks){
               resolve("This is true"); // then() will be called
          } else {
               reject("This is false"); // catch() will be called
          }
     })
}
giveMePizza()
        .then(data => console.log(data))
        .catch(err => console.error(err));
console.log('I will run immediately after calling giveMePizza() and
before any result arrives');
```

The callback from the `Promise` constructor gives us two parameters, `resolve` and `reject` functions, that will affect the state of the `Promise` object. If everything works, call `resolve`, otherwise call `reject`. Note that you can pass in values to `resolve` and `reject` which will be further passed on to the respective handlers, `then` and `catch`.

References

- https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise