# HTML5, CSS3 & JS

Chapter 5

Advanced JavaScript

# Chapter Objectives

In this chapter, we will discuss:

- JavaScript support of some functional programming features
- Several functions for working with arrays and lists
- How higher order functions accept functions as arguments and/or return a function
- Several frameworks that provide support for reactive programming in JavaScript

# Chapter Concepts

**Working with Arrays and Lists**

Higher Order Functions

Asynchronous JavaScript

Composition

Currying

Reactive Programming

Chapter Summary

# The `forEach()` Function

- To iterate over a collection of objects:
  - Can use a `for` loop
  - Or use the `forEach` function
- The `forEach` function is defined in Array.prototype
  - Requires a function that will execute for each object in the array

```
var products = [
  { name: 'Golf clubs', price: 175 },
  { name: 'Basketball',    price: 25 },
  { name: 'Tennis racket',   price: 95 },
  { name: 'Baseball glove', price: 65 },
  { name: 'Catnip mouse',  price: 5.5 }
];

// traditional for loop
for (var i = 0; i < products.length; i++) {
  console.log(products[i]);
}

// the forEach function
products.forEach(function(product, index) {
  console.log(product);
});
```

# Which to Use?

- The `forEach` function improves readability
  - The next object is automatically passed to the function argument
  - Don't have to use loop counter variables
- Fewer off-by-one errors with `forEach`
  - No loop counter variables
  - No off-by-one bug
- The for loop allows breaking out early
  - If you need to break out of the loop before iterating completely through it:
    - Use the `break` key word
    - Not available in `forEach`

# The `map()` Function

- The `map()` function is defined in Array.prototype
- Creates a new array
  - Containing the results of calling the provided function on each array element
- The function passed to map can take three arguments
  - The current object
  - The index of the current object
  - The array being processed
- The map function can take an optional second argument
  - The object to be used as **`this`**

```
// the map() function
var numbers = [1, 5, 10, 15];
var doubles = numbers.map(function(x) {
   return x * 2;
});
// doubles is now [2, 10, 20, 30]
// numbers is still [1, 5, 10, 15]
```

```
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]
```

# The `filter()` Function

- What if you only want to transform some of the values?
  - Map works on all the values
- The filter function solves this problem
  - It takes a function that returns a Boolean
    - True, keep the value
    - False, discard the value

```
var numbers = [1, 2, 3, 4];

var newNumbers = numbers.filter(function(number){
      return (number % 2 !== 0);
      }).map(function(number){
    return number * 2;
    });
```

# The `reduce()` Function

- What if I want to sum the values in an array?
  - Using the reduce function does the trick
- The second argument to reduce is the starting value
- The first argument to reduce is the function called for each element in the array

```
var numbers = [1, 2, 3, 4];

var totalNumber = numbers.map(function(number){
          return number * 2;
        }).reduce(function(total, number){
            return total + number;
        }, 0);

console.log("The total number is", totalNumber); // 20
```

# Exercise 5.1: Working with Arrays


**20 min**

- View the files in the Ch05\workingWithArrays folder.
- View the `workingWithArrays.html` file in your favorite browser.
- Run the code by clicking the different buttons on the page.
- Complete the code in the workingWithArraysAndLists.js file.
- View the web page again in your browser.
- Verify that the code you added works correctly.

# Chapter Concepts

- Working with Arrays and Lists
- **Higher Order Functions**
- Asynchronous JavaScript
- Composition
- Currying
- Reactive Programming
- Chapter Summary

# Higher Order Functions

- A higher order function is a function that takes another function as an argument
  - Like map, filter, reduce, etc.
  - The argument function is often referred to as a callback
- A function that returns a function:
  - Is also known as a higher order function

# No More Loops

- Using higher order functions
  - There is no need for most imperative loops
  - No need to use the for loop
- The map, filter, reduce functions
  - Apply an argument function to every element in a collection
- Side-effect-free
  - Array higher order functions do not mutate the array they are called on

# Optional Exercise 5.2: Using Higher Order Functionss


**20 min**

- View the files in the Ch05\higherOrderFunctions folder.
- View the `higherOrderFunctions.html` file in your favorite browser.
- Run the code by clicking the different buttons on the page.
- Complete the code in the `higherOrderFunctions.js` file.
- View the web page again in your browser.
- Verify that the code you added works correctly.

# Chapter Concepts

- Working with Arrays and Lists
- Higher Order Functions
- **Asynchronous JavaScript**
- Composition
- Currying
- Reactive Programming
- Chapter Summary

# Asynchronous Function Calls

- Asynchronous programming in JavaScript started with callbacks
  - Pass a function as an argument to another function
  - That function calls the callback function when it is finished with its job
- Problems with callbacks
  - “Callback Hell”
    - Easy to write spaghetti code
  - Easy to miss error handling
  - Can’t return values with the return key word
- Next came Promises

```
Something.doIt(function(err) {
  if (err)  {
    //error handling
    return;
  }
  console.log('success');
});
```

# Promises

- A Promise represents the eventual result of an asynchronous operation
- Promises use callbacks
  - Both then and catch register callbacks
  - Invoked with the result of the async operations
    - Or the reason it could not be fulfilled
- A Promise can be:
  - Fulfilled – the action succeeded
  - Rejected – the action failed
  - Pending – the action is not yet complete
  - Settled – the action has completed

```
const aVerySimplePromise = new Promise((resolve, reject) => {
 // fulfilled
  if(conditionIsTrue) resolve(someValue);
 // rejected
  else reject("failure reason");
});
```

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

# Promises Can Be Chained

- Promises can be chained
  - Each link in the chain registers a callback
  - And returns another Promise
- Much easier to read than “Callback Hell”

```
saveSomething()
.then(updateSomethingImportant)
.then(deleteBogusStuff)
.then(logResults)
.catch(logErrors);
```

# Optional Exercise 5.3: Using JavaScript Promises


**20 min**

- View the files in the Ch05\promises folder.
- View the `promises.html` file in a local web server.
  - a. Right-click the `promises.html` file in the Explorer window.
  - b. Choose **Open with Live Server**.
  - c. The `promises.html` file will be displayed in your default browser hosted on localhost (127.0.0.1:5500).
- Run the code by clicking the different buttons on the page.
- Complete the code in the `promises.js` file.
- View the web page again in the local web server.
- Verify that the code you added works correctly.

# Chapter Concepts

- Working with Arrays and Lists
- Higher Order Functions
- Asynchronous JavaScript
- **Composition**
- Currying
- Reactive Programming
- Chapter Summary

# Function Composition

- Function composition is the process of combining two or more functions
  - To produce a new function:
  - `compose(f, g)(x) equals f(g(x))`

```
var compose = function(f, g) {
  return function(x) {
    return g(f(x));
  }
};
```

- This can be done more concisely with a lambda expression

```
var lambdaCompose = (f,g) => (x) => g(f(x));
```

# Function Composition (continued)

- What will this produce?

```
var trim = function(str) {return str.replace(/^\s*|\s*$/g, '');};

var capitalize = function(str) {return str.toUpperCase();};

var convert = compose(trim, capitalize);

console.log('"' + convert('   abc def ghi  ') + '"');
```

- What will this produce?

```
var add1 = function(x) {return x + 1;};
var mult2 = function(x) {return x * 2;};

// what will this produce?
var f = compose(add1, square);
console.log(f(7));
```

# Exercise 5.4: Using Function Composition


**20 min**

- View the files in the Ch05\functionComposition folder.
- View the functionComposition.html file in a local web server.
  - Right-click the `functionComposition.html` file in the Explorer window.
  - Choose **Open with Live Server**.
  - The `functionComposition.html` file will be displayed in your default browser hosted on localhost (127.0.0.1:5500).
- Run the code by clicking the different buttons on the page.
- Complete the code in the `functionComposition.js` file.
- View the web page again in the local web server.
- Verify that the code you added works correctly.

# Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Asynchronous JavaScript

Composition

**Currying**

Reactive Programming

Chapter Summary

# Currying

- Currying is a way to construct functions that allow partial application of the functions arguments
  - If you pass all the arguments in a call:
    - You get the result back
  - If you pass a subset of the arguments:
    - You get a function that expects the rest of the arguments

```
function multiply(x, y) { return x * y; }

function curriedMultiply(x) {
  return function(y) { return x * y; }
}
```

```
var multiplyBy5 =
curriedMultiply(5);
multiply(5,2) ===
multiplyBy5(2);
```

# Currying and Callbacks

- Currying can be combined with callbacks
  - To create a higher order *factory* function
- Useful in event handling
- Can replace the callback pattern used in node.js
- This code is an example of combining currying with node.js
  - To process a file
  - Allows for the read data to be passed around
  - As the file is being processed
  - Defer invoking the read function's callback
    - Until the result is needed
  - Can allow for sequential and parallel i/o processing of multiple files

```
// a curried version of node's fs.readFile(path,
encoding, callback) function
var readFile = curriedReadFile(path);

readFile(function(err, data) {
  if (err) {
    throw err;
  }
  // do something clever with the data
});
```

# The Power of Currying

- With currying, you can separate the initiation of an asynchronous operation
  - From the retrieval of the result
- So, it is possible to initiate several operations in close sequence
  - Let them do their i/o in parallel
  - Retrieve the results later

```
var reader1 = curriedReadFile(path1, "utf8");
var reader2 = curriedReadFile(path2, "utf8");
// I/O is parallelized and we can do other important
things while it runs

// further down the line:
reader1(function(err, data1) {
  reader2(function(err, data2) {
    // do something clever with data1 and data2
  });
});
```

# Optional Exercise 5.5: Currying


**20 min**

- View the files in the Ch05\currying folder.
- View the `currying.html` file in a local web server.
  - a. Right-click the `currying.html` file in the Explorer window.
  - b. Choose **Open with Live Server**.
  - c. The `currying.html` file will be displayed in your default browser hosted on localhost (127.0.0.1:5500).
- Run the code by clicking the different buttons on the page.
- Complete the code in the `currying.js` file.
- View the web page again in the local web server.
- Verify that the code you added works correctly.

# Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Asynchronous JavaScript

Composition

Currying

**Reactive Programming**

Chapter Summary

# Reactive Programming

- Reactive systems are:
  - Responsive
    - Responds in a timely manner
    - Focus on providing rapid and consistent response times
  - Resilient
    - Stays responsive in the face of failure
  - Elastic
    - Stays responsive under varying workload
  - Message-driven
    - Relies upon asynchronous message passing

From *The Reactive Manifesto*, http://www.reactivemanifesto.org/

# From Promises to Observables

- Promises act on data
  - And then return a single value
- Observables are the observer pattern at work
  - Can produce multiple values asynchronously over time
- An observable is a function that takes an observer
  - And returns a cancellation function
- An observer is an object with next, error, and complete functions

# The Rx Library

- Reactive Extensions (RxJs)
  - A library for composing asynchronous applications
  - Using observable sequences and LINQ-style query operators
  - Works with synchronous and asynchronous data streams

| | Single Return Value | Multiple Return Values |
|---|---|---|
| Pull/Synchronous/Interactive | Object | Iterables(Array…) |
| Push/Asynchronous/Reactive | Promise | Observable |

LINQ – Language Integrated Query

# Reactive Programming in JavaScript

- Reactive programming revolves around asynchronous data
  - Called observables
  - Or streams
- There are quite a few choices for frameworks that extend JavaScript
  - And add support for reactive programming
  - Reactive Extension (RxJs)
  - ReactiveX
  - Omniscient
  - WebRx

# Observable Example

- The basic building blocks of RxJs
  - Observables (producers)
  - Observers (consumers)
- Two types of observables
  - Hot observables
    - Pushing even if we are not subscribed to them
  - Cold observables
    - Pushing only when we subscribe to them

```
// Creates an observable sequence of 5
// integers, starting from 1
var source = Rx.Observable.range(1, 5);

// Prints out each item
var subscription = source.subscribe(
function (x) { console.log('onNext: %s', x); },
function (e) { console.log('onError: %s', e); },
function () { console.log('onCompleted'); });

// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 4
// => onNext: 5
// => onCompleted
```

# Prime Number Generator Example

- Suppose we want to aggregate the results from a prime number generator over time
  - So the user interface does not have to deal with too many updates
  - We are interested in only the number of generated prime numbers
- Probably want to use a buffer
  - RxJs provides a buffer function
- Also, may want to use a map
  - To transform the data
- The `fromEvent` function constructs an observable

```
var worker = new Worker('prime.js');
var observable = Rx.Observable.fromEvent(worker, 'message')
                    .map(function (ev) { return ev.data * 1; })
                    .buffer(Rx.Observable.interval(500))
                    .where(function (x) { return x.length > 0; })
                    .map(function (x) { return x.length; });
```

# Exercise 5.6: Using Observables


**20 min**

- View the files in the Ch05\observables folder.
- View the `observables.html` file in a local web server.
  - a. Right-click the `observables.html` file in the Explorer window.
  - b. Choose **Open with Live Server**.
  - c. The `observables.html` file will be displayed in your default browser hosted on localhost (127.0.0.1:5500).
- Run the code by clicking the different buttons on the page.
- Complete the code in the `observables.js` file.
- View the web page again in the local web server.
- Verify that the code you added works correctly.

# Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Asynchronous JavaScript

Composition

Currying

Reactive Programming

**Chapter Summary**

# Chapter Summary

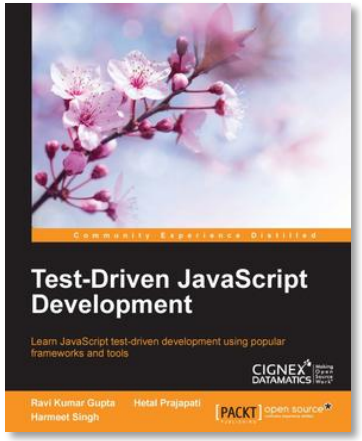
In this chapter, we have discussed:

- JavaScript support of some functional programming features
- Several functions for working with arrays and lists
- How higher order functions accept functions as arguments and/or return a function
- Several frameworks that provide support for reactive programming in JavaScript

# Resources

- There is much more to the testing JavaScript
- Here are some resources to aid in further explorations


- JavaScript: The Good Parts
  - Douglas Crockford
  - https://www.youtube.com/watch?v=hQVTIJBZook
  - Available on Safari Books Online




- *Test-Driven JavaScript Development*
  - Gupta, Prajapati & Singh, Packt Publishing
  - Available on Safari Books Online