# Programming JavaScript Applications

## Chapter 1. The JavaScript Revolution

JavaScript, HTML, and CSS have become so prevalent that many operating systems have adopted the open web standards as the presentation layer for native apps. iPhone and Android mobile devices support web views that allow them to incorporate JavaScript and HTML5 functionality into native applications.

### Advantages of JavaScript

With a familiar C-like syntax, JS is one of the most advanced and expressive programming languages with encapsulation, polymorphism, multiple inheritance, composition, just-in-time compiling, garbage collection, dynamic binding.

Performance - JavaScript apps are event driven and nonblocking which makes them well suited for a presentation layer.

JSON provides easy data serialization.

Prototypical inheritance means that objects inherit methods through their prototype chain and you can modify an object’s prototype on the fly. It is possible to mimic class-based inheritance in JS.

First-Class Functions - nearly everything in JavaScript is an object, including functions. Functions can be used as variables and anonymous functions are easy to use.

A functional programming style is used to abstract commonly repeating coding patterns like .map(), .reduce() and .forEach(). Underscore.js contains many functional utlitities.

JSON has become the most popular document format for RESTful Web Services.

## Chapter 2. Functions

Guidelines: Don’t Repeat Yourself (DRY), Do One Thing (DOT), Keep It Simple Stupid (KISS) & Less Is More

Two common bugs: syntax errors and unintentional side-effects (the bane of code re-use)

A pure function does not alter existing variables or program state in any way, and always returns the same value given the same inputs. You can still alter program state. REST works this way: you get a copy of the data resource (called a representation), manipulate it, and send the copy back to the sender.

Writing most of your functions in a similar fashion can help you separate concerns and reduce code duplication.

There are several ways to define functions in JavaScript:

* Function Declaration  
  function foo() {}
* Anonymous Function Expression  
  var bar = function () {}
* Named Function Expression  
  var bar = function bar() {}
* Anonymous Method Literal (Function Expression)  
  var bar = { foo: function () {} };
* Named Method Literal (Function Expression)  
  var bar = { foo: function foo() {} };

Adding a name to a function expression is useful for recursion and debugging but you cannot use the function name outside of the function, you have to use the variable name. Some older browsers (e.g. IE8) got this wrong so you might have conflicts between function names in expressions and other variable names.

A lambda expression is an anonymous function that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for manipulating collections or as event and callback handlers.

A language that supports first class functions supports:

* passing functions as arguments to other functions,
* returning them as the values from other functions, and;
* assigning them to variables or data structures

Accessing variables outside of the immediate lexical scope creates a closure. In other words, a closure is formed when a nested function is defined inside of another function, allowing access to the outer functions variables. A closure also stores function state, even after the function has returned

Immediately Invoked Function Expressions - It’s possible in JavaScript to immediately invoke a function as soon as it’s defined. This technique is often used to create a new scope to encapsulate modules.

The .call() and .apply() methods allows you to call any method or function on any object (functional polymorphism):

* someMethod.call(context, argument1, argument2, ...);
* someMethod.apply(context, someArray);

Hoisting is the word most commonly used to describe the illusion that all variable declarations are “hoisted” to the top of the containing function.

It is important to understand these concepts before you begin to approach advanced design patterns, as scope and context play a fundamental role in modern JavaScript.

Named parameters are possible by using extension methods commonly found in libraries like jQuery and underscore.

Generic programming is a style that attempts to express algorithms and data structures in a way that is type agnostic. Generic programming is particularly relevant to functional programming.

Method chaining is using the output of one method call as the context of the next method call.

A fluent API is one that reads like natural language. One of the primary benefits of method chaining is that it can be used to support fluent APIs.

Functional programming is a style of programming that uses higher-order functions (as opposed to objects and data) to facilitate code organization and reuse. A higher order function treats functions as data, either taking a function as an argument or returning a function as a result.

To maximize code reuse, try to make as many functions as possible both stateless and generic (or polymorphic).

Asynchronous operations are operations that happen outside the linear flow of program execution. Asynchronous operations are broken up into two phases: call and response or register/attach/add and callback.

The difference between a promise and a callback is that a promise is an object that gets returned from the callee, instead of a function that gets passed into and invoked by the callee.

A deferred is the object that controls the promise, with a few extra methods.

## Chapter 3. Objects

JavaScript is not a classical OO language. It’s a prototypal language.

In Design Patterns: Elements of Reusable Object Oriented Software, the Gang of Four opened the book with two foundational principles of object-oriented design:

* Program to an interface, not an implementation.
* Favor object composition over class inheritance.

JavaScript can do most of the OO things you’re accustomed to in other languages; however, JavaScript has many native capabilities that make some classical OO features and patterns obsolete.

JavaScript AntiPattern: Constructors exist but are unnecessary in JS and they make you think of classes and inheritance. Dynamic object extension and prototypal inheritance are a much more powerful and flexible way to reuse code.

JavaScript Strengths:

* Lambdas and closures
* Object-literal notation
* Dynamic object extension
* Prototypes
* Factories
* Fluent APIs

Weirdness about prototype inheritance:

Properties on the prototype act are copied to the instances with a shallow copy. If you change the top-level properties, including a pointer, it only affects the instance. If the top-level properties are objects or arrays and you change their contents it has the potential to affect the prototype and some unknown number of other instances.

It’s important to note though that if you mutate an object or array property on the prototype, that mutation will be shared on the prototype. If you replace the whole property, the change is reflected only on that instance: