CHAPTER 1

Expressive JavaScript

JavaScript is one of the most popular and widely used languages in the world today. Because it is embedded in all modern browsers, it has an extraordinarily wide distribution. As a language, it is incredibly important in our daily lives, powering the websites that we go to and helping the Web to present a rich interface.

Why then do some still consider it to be a toy language, not worthy of the professional programmer? We think it is because people do not realize the full power of the language and how unique it is in the programming world today. JavaScript is a very expressive language, with some features that are uncommon to the C family of languages.

In this chapter we explore some of the features that make JavaScript so expressive. We look at how the language allows you to accomplish the same task in a number of different ways and how you can take alternative approaches to object-oriented programming by using concepts from functional programming. We discuss why you should use design patterns in the first place and how adapting them to JavaScript will make your code more efficient and easier to work with.

The Flexibility of JavaScript

One of the most powerful features of the language is its flexibility. As a JavaScript programmer, you can make your programs as simple or as complex as you wish them to be. The language also allows several different programming styles. You can write your code in the functional style or in the slightly more complex object-oriented style. It also lets you write relatively complex programs without knowing anything at all about functional or object-oriented programming; you can be productive in this language just by writing simple functions. This may be one of the reasons that some people see JavaScript as a toy, but we see it as a good thing. It allows programmers to accomplish useful tasks with a very small, easy-to-learn subset of the language. It also means that JavaScript scales up as you become a more advanced programmer.

JavaScript allows you to emulate patterns and idioms found in other languages. It even creates a few of its own. It provides all the same object-oriented features as the more traditional server-side languages.

Let's take a quick look at a few different ways you can organize code to accomplish one task: starting and stopping an animation. It's OK if you don't understand these examples; all of the patterns and techniques we use here are explained throughout the book. For now, you can view this section as a practical example of the different ways a task can be accomplished in JavaScript.

If you're coming from a procedural background, you might just do the following:

```
/* Start and stop animations using functions. */

function startAnimation() {
  ...
}

function stopAnimation() {
  ...
}
```

This approach is very simple, but it doesn't allow you to create animation objects, which can store state and have methods that act only on this internal state. This next piece of code defines a class that lets you create such objects:

```
/* Anim class. */

var Anim = function() {
  ...
};
Anim.prototype.start = function() {
  ...
};
Anim.prototype.stop = function() {
  ...
};

/* Usage. */

var myAnim = new Anim();
myAnim.start();
...
myAnim.stop();
```

This defines a new class called `Anim` and assigns two methods to the class's `prototype` property. We cover this technique in detail in Chapter 3. If you prefer to create classes encapsulated in one declaration, you might instead write the following:

```
/* Anim class, with a slightly different syntax for declaring methods. */

var Anim = function() {
  ...
};
Anim.prototype = {
  start: function() {
    ...
  },
```

```
  stop: function() {
    ...
  }
};
```

This may look a little more familiar to classical object-oriented programmers who are used to seeing a class declaration with the method declarations nested within it. If you've used this style before, you might want to give this next example a try. Again, don't worry if there are parts of the code you don't understand:

```
/* Add a method to the Function object that can be used to declare methods. */

Function.prototype.method = function(name, fn) {
  this.prototype[name] = fn;
};

/* Anim class, with methods created using a convenience method. */

var Anim = function() {
  ...
};
Anim.method('start', function() {
  ...
});
Anim.method('stop', function() {
  ...
});
```

`Function.prototype.method` allows you to add new methods to classes. It takes two arguments. The first is a string to use as the name of the new method, and the second is a function that will be added under that name.

You can take this a step further by modifying `Function.prototype.method` to allow it to be chained. To do this, you simply `return this` after creating each method. We devote Chapter 6 to chaining:

```
/* This version allows the calls to be chained. */

Function.prototype.method = function(name, fn) {
    this.prototype[name] = fn;
    return this;
};

/* Anim class, with methods created using a convenience method and chaining. */

var Anim = function() {
  ...
};
```

```
Anim.
  method('start', function() {
    ...
  }).
  method('stop', function() {
    ...
  });
```

You have just seen five different ways to accomplish the same task, each using a slightly different style. Depending on your background, you may find one more appealing than another. This is fine; JavaScript allows you to work in the style that is most appropriate for the project at hand. Each style has different characteristics with respect to code size, efficiency, and performance. We cover all of these styles in Part 1 of this book.

A Loosely Typed Language

In JavaScript, you do not declare a type when defining a variable. However, this does not mean that variables are not typed. Depending on what data it contains, a variable can have one of several types. There are three primitive types: booleans, numbers, and strings (JavaScript differs from most other mainstream languages in that it treats integers and floats as the same type). There are functions, which contain executable code. There are objects, which are composite datatypes (an array is a specialized object, which contains an ordered collection of values). Lastly, there are the `null` and `undefined` datatypes. Primitive datatypes are passed by value, while all other datatypes are passed by reference. This can cause some unexpected side effects if you aren't aware of it.

As in other loosely typed languages, a variable can change its type, depending on what value is assigned to it. The primitive datatypes can also be cast from one type to another. The `toString` method converts a number or boolean to a string. The `parseFloat` and `parseInt` functions convert strings to numbers. Double negation casts a string or a number to a boolean:

```
var bool = !!num;
```

Loosely typed variables provide a great deal of flexibility. Because JavaScript converts type as needed, for the most part, you won't have to worry about type errors.

Functions As First-Class Objects

In JavaScript, functions are first-class objects. They can be stored in variables, passed into other functions as arguments, passed out of functions as return values, and constructed at run-time. These features provide a great deal of flexibility and expressiveness when dealing with functions. As you will see throughout the book, these features are the foundation around which you will build a classically object-oriented framework.

You can create *anonymous functions*, which are functions created using the `function() { ... }` syntax. They are not given names, but they can be assigned to variables. Here is an example of an anonymous function:

```
/* An anonymous function, executed immediately. */

(function() {
  var foo = 10;
  var bar = 2;
  alert(foo * bar);
})();
```

This function is defined and executed without ever being assigned to a variable. The pair of parentheses at the end of the declaration execute the function immediately. They are empty here, but that doesn't have to be the case:

```
/* An anonymous function with arguments. */

(function(foo, bar) {
  alert(foo * bar);
})(10, 2);
```

This anonymous function is equivalent to the first one. Instead of using `var` to declare the inner variables, you can pass them in as arguments. You can also return a value from this function. This value can be assigned to a variable:

```
/* An anonymous function that returns a value. */

var baz = (function(foo, bar) {
  return foo * bar;
})(10, 2);

// baz will equal 20.
```

The most interesting use of the anonymous function is to create a closure. A *closure* is a protected variable space, created by using nested functions. JavaScript has function-level scope. This means that a variable defined within a function is not accessible outside of it. JavaScript is also lexically scoped, which means that functions run in the scope they are defined in, not the scope they are executed in. These two facts can be combined to allow you to protect variables by wrapping them in an anonymous function. You can use this to create private variables for classes:

```
/* An anonymous function used as a closure. */

var baz;

(function() {
  var foo = 10;
  var bar = 2;
  baz = function() {
    return foo * bar;
  };
})();
```

```
baz(); // baz can access foo and bar, even though it is executed outside of the
       // anonymous function.
```

The variables `foo` and `bar` are defined only within the anonymous function. Because the function `baz` was defined within that closure, it will have access to those two variables, even after the closure has finished executing. This is a complex topic, and one that we touch upon throughout the book. We explain this technique in much greater detail in Chapter 3, when we discuss encapsulation.

The Mutability of Objects

In JavaScript, everything is an object (except for the three primitive datatypes, and even they are automatically wrapped with objects when needed). Furthermore, all objects are *mutable*. These two facts mean you can use some techniques that wouldn't be allowed in most other languages, such as giving attributes to functions:

```
function displayError(message) {
  displayError.numTimesExecuted++;
  alert(message);
};
displayError.numTimesExecuted = 0;
```

It also means you can modify classes after they have been defined and objects after they have been instantiated:

```
/* Class Person. */

function Person(name, age) {
  this.name = name;
  this.age = age;
}
Person.prototype = {
  getName: function() {
    return this.name;
  },
  getAge: function() {
    return this.age;
  }
}

/* Instantiate the class. */

var alice = new Person('Alice', 93);
var bill = new Person('Bill', 30);

/* Modify the class. */
```

```
Person.prototype.getGreeting = function() {
  return 'Hi ' + this.getName() + '!';
};

/* Modify a specific instance. */

alice.displayGreeting = function() {
  alert(this.getGreeting());
}
```

In this example, the `getGreeting` method is added to the class after the two instances are created, but these two instances still get the method, due to the way the `prototype` object works. `alice` also gets the `displayGreeting` method, but no other instance does.

Related to object mutability is the concept of *introspection*. You can examine any object at run-time to see what attributes and methods it contains. You can also use this information to instantiate classes and execute methods dynamically, without knowing their names at development time (this is known as *reflection*). These are important techniques for dynamic scripting and are features that static languages (such as C++) lack.

Most of the techniques that we use in this book to emulate traditional object-oriented features rely on object mutability and reflection. It may be strange to see this if you are used to languages like C++ or Java, where an object can't be extended once it is instantiated and classes can't be modified after they are declared. In JavaScript, everything can be modified at run-time. This is an enormously powerful tool and allows you to do things that are not possible in those other languages. It does have a downside, though. It isn't possible to define a class with a particular set of methods and be sure that those methods are still intact later on. This is part of the reason why type checking is done so rarely in JavaScript. We cover this in Chapter 2 when we talk about duck typing and interface checking.

Inheritance

Inheritance is not as straightforward in JavaScript as in other object-oriented languages. JavaScript uses object-based (prototypal) inheritance; this can be used to emulate class-based (classical) inheritance. You can use either style in your code, and we cover both styles in this book. Often one of the two will better suit the particular task at hand. Each style also has different performance characteristics, which can be an important factor in deciding which to use. This is a complex topic, and we devote Chapter 4 to it.

Design Patterns in JavaScript

In 1995, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides published a book titled *Design Patterns*. This book catalogs the different ways objects can interact with each other and it created a common vocabulary around the different types of objects. The blueprints for creating these different types of objects are called *design patterns*. The book describes these patterns in a somewhat language-agnostic way, so that they can be used anywhere. The book you are holding in your hands takes those patterns and applies them specifically to JavaScript.

The fact that JavaScript is so expressive allows you to be very creative in how design patterns are applied to your code. There are three main reasons why you would want to use design patterns in JavaScript:

1. *Maintainability*: Design patterns help to keep your modules more loosely coupled. This makes it easier to refactor your code and swap out different modules. It also makes it easier to work in large teams and to collaborate with other programmers.

2. *Communication*: Design patterns provide a common vocabulary for dealing with different types of objects. They give programmers shorthand for describing how their systems work. Instead of long explanations, you can just say, "It uses the factory pattern." The fact that a particular pattern has a name means you can discuss it at a high level, without having to get into the details.

3. *Performance*: Some of the patterns we cover in this book are optimization patterns. They can drastically improve the speed at which your program runs and reduce the amount of code you need to transmit to the client. The flyweight (Chapter 13) and proxy (Chapter 14) patterns are the most important examples of this.

There are two reasons why you might *not* want to use design patterns:

1. *Complexity*: Maintainability often comes at a cost, and that cost is that your code may be more complex and less likely to be understood by novice programmers.

2. *Performance*: While some patterns improve performance, most of them add a slight performance overhead to your code. Depending on the specific demands of your project, this overhead may range from unnoticeable to completely unacceptable.

Implementing patterns is the easy part; knowing which one to use (and when) is the hard part. Applying design patterns to your code without knowing the specific reasons for doing so can be dangerous. Make an effort to ensure that the pattern you select is the most appropriate and won't degrade performance below acceptable limits.

Summary

The expressiveness of JavaScript provides an enormous amount of power. Even though the language lacks certain useful built-in features, its flexibility allows you to add them yourself. You can write code to accomplish a task in many different ways, depending on your background and personal preferences.

JavaScript is loosely typed; programmers do not declare a type when defining a variable. Functions are first-class objects and can be created dynamically, which allows you to create closures. All objects and classes are mutable and can be modified at run-time. There are two styles of inheritance you can use, prototypal and classical, and each has its own strengths and weaknesses.

Design patterns in JavaScript can be extremely helpful and beneficial, but they can also be detrimental if used improperly. In a language as lightweight as JavaScript, overly complex architectures can quickly bog down your application. Always make sure the style of programming you use and the patterns you select are right for the job.