**Object-Oriented Programming**

Before diving into JavaScript let's take a moment to review what people mean when they say "object-oriented", and what the main features of this programming style are. Here's a list of concepts that are most often used when talking about object-oriented programming (OOP):

Object, method, property

Class

Encapsulation

Aggregation

Reusability/inheritance

Polymorphism

Let's take a closer look into each one of these concepts.

**Objects**

As the name *object-oriented* suggests, objects are quite important. An object is a representation of a "thing" (someone or something), and this representation is expressed with the help of a programming language. The thing can be anything—a real-life object, or some more convoluted concept. Taking a common object like a cat for example, you can see that it has certain characteristics (color, name, weight) and can perform some actions (meow, sleep, hide, escape). The characteristics of the object are called *properties* in OOP and the actions are called *methods*.

There is also an analogy with the spoken language:

Objects are most often named using nouns (book, person)

Methods are verbs (read, run)

Values of the properties are adjectives

**Classes**

In real life, similar objects can be grouped based on some criteria. A hummingbird and an eagle are both birds, so they can be classified as belonging to the Birds class. In OOP, a class is a blueprint, or recipe for an object. Another name for "object" is "instance", so we say that the eagle is an instance of the Birds class. You can create different objects using the same class, because a class is just a template, while the objects are concrete instances, based on the template. JavaScript there are no classes; everything is based on objects. JavaScript has the notion of prototypes, which are also objects (we'll discuss them later in detail). In a classic OO language, you'd say something like "create me a new object called Bob which is of class Person

**Encapsulation**

*Encapsulation* is another OOP-related concept, which illustrates the fact that an object contains (encapsulates) both:

Data (stored in properties) and

The means to do something with the data (using methods)

One other term that goes together with encapsulation is *information hiding*. This is a rather broad term and can mean different things, but let's see what people usually mean when they use it in the context of OOP.

Imagine an object, say an MP3 player. You, as a user of the object, are given some interface to work with, such as buttons, the display, and so on. You use the interface in order to get the object to do something useful for you, like playing a song. Exactly how it is working on the inside, you don't know and, most often, don't care. In other words, the implementation of the interface is hidden from you. The same thing happens in OOP, when your code uses an object by calling its methods. It doesn't matter if you coded the object yourself or it came from some third party library; your code doesn't need to know how the methods work internally. In compiled languages, you can't actually read the code that makes an object work. In JavaScript, because it's an interpreted language, you can see the source code, but the concept is still the same—you work with the object's interface, without worrying about its implementation.

Another aspect of information hiding is the visibility of methods and properties. In some languages, objects can have *public*, *private*, and *protected* methods and properties. This categorization defines the level of access the users of the object have. For example, only the internal implementation of the object has access to the private methods, while anyone has access to the public ones. In JavaScript, all methods and properties are public, but we'll see that there are ways to protect the data inside an object and achieve privacy.

**Inheritance**

Inheritance is a very elegant way to reuse code that has already been written. For example, you can have a generic object Person, which has properties such as name and date of birth, and that implements the functionality walk, talk, sleep, eat. Then you figure out that you need an object Programmer. You could re-implement all the methods and properties that Person has, but it would be smarter to just say that Programmer *inherits* Person, and save yourself some work. The Programmer object only needs to implement more-specific functionality, such as the method "write code", while reusing all of the Person's functionality.

In classical OOP, classes inherit from other classes, but in JavaScript, because there are no classes, objects inherit from other objects.

When an object inherits from another object, it usually adds new methods to the inherited ones, thus *extending* the old object. Often the following phrases can be used interchangeably:

**Polymorphism**

In the example above, we had a Programmer object that inherited all of the methods of the parent Person object. This means that both objects provide a "talk" method, among others. Now imagine that somewhere in our code, there's a variable called Bob and it so happens that we don't know if Bob is a Person, or a Programmer object. We can still call the "talk" method on the Bob object and the code will work. This ability to call the same method on different objects and have each of them respond in their own way is called *polymorphism*

Primitive Data Types, Arrays, Loops, and Conditions

Before diving into the object-oriented features of JavaScript, let's first take a look at some of the basics. This chapter walks you through:

The primitive data types in JavaScript, such as strings and numbers

Arrays

Common operators, such as +, -, delete, and typeof

Flow control statements, such as loops and if-else conditions

**Variables**

Variables are used to store data. When writing programs, it is convenient to use variables instead of the actual data, as it's much easier to write pi instead of 3.141592653589793 especially when it happens several times inside your program. The data stored in a variable can be changed after it was initially assigned, hence the name "variable".

To *initialize* a variable means to give it a value for the first (initial) time. You have two ways to do so:

Declare the variable first, then initialize it, or

Declare and initialize with a single statement

**Operators**

Operators take one or two values (or variables), perform an operation, and return a value.

**Primitive Data Types**

Any value that you use is of a certain *type*. In JavaScript, there are the following *primitive* data types:

1. Number—this includes floating point numbers as well as integers, for example 1, 100, 3.14.

2. String—any number of characters, for example "a", "one", "one 2 three".

3. Boolean—can be either true or false.

4. Undefined—when you have declared a variable, but not given it a value yet. JavaScript will initialize it behind the scenes, with the value undefined.

5. Null—this is another special data type that can have only one value, the null value. It means no value, an empty value, nothing. The difference with undefined is that if a variable has a value null, it is still defined, it only happens that its value is nothing. You'll see some examples shortly.

**Logical Operators**

There are three operators, called *logical operators,* that work with boolean values. These are:

!—logical NOT (negation)

&&—logical AND

||—logical OR

**Arrays of arrays**

An array can contain any type of values, including other arrays.

**[1, "two", false, null, undefined, [1, 2, 3]]**

There are more ways to have fun with arrays (and we'll get to that in Chapter 4), but let's stop here for now, remembering that:

An array is a data store

An array contains indexed elements

Indexes start from zero and increment by one for each element in the array

To access array elements we use the index in square brackets

An array can contain any type of data, including other arrays.

**Conditions and Loops**

*Conditions* provide a simple but powerful way to control the flow of execution through a piece of code. *Loops* allow you to perform repeating operations with less code. Let's take a look at:

if conditions,

switch statements,

while, do-while, for, and for-in loops.

**Switch**

If you find yourself using an if condition and having too many else if parts, you could consider changing the if to a switch.

An *infinite loop* is when the condition is always true and your code gets stuck in the loop "forever". This is, of course, is a logical error and you should look out for such scenarios.

In JavaScript, there are four types of loops:

while loops

do-while loops

for loops

for-in loops

while loops are the simplest type of loop. They look like this:

var i = 0;

while (i < 10) {

i++;

}

The while statement is followed by a condition in parentheses and a code block in curly brackets. As long as the condition evaluates to true, the code block is executed over and over again.

**For Loops**

for is the most widely used type of loop and you should make sure you're comfortable with this one. It requires a just little bit more in terms of syntax.

**What is a Function?**

Functions allow you group together some code, give this code a name, and reuse it later, addressing it by name. Let's see an example:

function sum(a, b) {

var c = a + b;

return c;

}

Understanding these topics will provide a solid base that will allow you to dive into the second part of the chapter, which shows some interesting applications of functions:

Using anonymous functions

Callbacks

Self-invoking functions

Inner functions (functions defined inside functions)

Functions that return functions

Functions that redefine themselves

Closures

Note that a function can only return a single value. If you need to return more values, then simply return an array that contains all of the values as elements of this array.

**Calling a Function**

In order to make use of a function, you need to *call* it. You call a function simply by using its name followed by any parameters in parentheses. "To *invoke*" a function is another way of saying "to call".

Let's call the function sum(), passing two parameters and assigning the value that the function returns to the variable result:

>>> var result = sum(1, 2);

>>> result;

**3**

**Pre-defined Functions**

There are a number of functions that are built into the JavaScript engine and available for you to use. Let's take a look at them. While doing so, you'll have a chance to experiment with functions, their parameters and return values , and become comfortable in working with them. The list of the built-in functions is:

parseInt()

parseFloat()

isNaN()

isFinite()

encodeURI()

decodeURI()

encodeURIComponent()

decodeURIComponent()

eval()

The parseInt() function parses a string and returns an integer.

The parseFloat() function parses a string and returns a floating point number.

The isNaN() function returns true if the value is NaN (Not-a-Number), and false if not.

**Anonymous Functions**

"An anonymous function is a function without a name."

This code may look a little odd, because it doesn't actually do anything, but the code is valid and is not going to cause an error. You can say that this code contains *anonymous* data—anonymous because the data pieces are not assigned to any variable and therefore don't have a name.

Now, these anonymous pieces of data scattered around your code are not really useful, except if they happen to be functions. In this case, there can be two elegant uses for them:

You can pass an anonymous function as a parameter to another function. The receiving function can do something useful with the function that you pass.

You can define an anonymous function and execute it right away.

This is an example of a function:

function helloWorld() {

alert('Hello World');

}

This is an example of an anonymous function:

var helloWold = function() {

alert('Hello World');

}

**Callback Functions:**

 A **JavaScript Callback Function** is a **function** that is passed as a parameter to another **JavaScript function**, and the**callback function** is run inside of the **function** it was passed into