Invocation

Invoking a function suspends the execution of the current function, passing control and parameters to the new function. In addition to the declared parameters, every function receives two additional parameters: `this` and `arguments`. The `this` parameter is very important in object oriented programming, and its value is determined by the *invocation pattern*. There are four patterns of invocation in JavaScript: the method invocation pattern, the function invocation pattern, the constructor invocation pattern, and the apply invocation pattern. The patterns differ in how the bonus parameter `this` is initialized.

The invocation operator is a pair of parentheses that follow any expression that produces a function value. The parentheses can contain zero or more expressions, separated by commas. Each expression produces one argument value. Each of the argument values will be assigned to the function's parameter names. There is no runtime error when the number of arguments and the number of parameters do not match. If there are too many argument values, the extra argument values will be ignored. If there are too few argument values, the `undefined` value will be substituted for the missing values. There is no type checking on the argument values: any type of value can be passed to any parameter.

The Method Invocation Pattern

When a function is stored as a property of an object, we call it a *method*. When a method is invoked, `this` is bound to that object. If an invocation expression contains a refinement (that is, a `.` dot expression or `[subscript]` expression), it is invoked as a method:

```
// Create myObject. It has a value and an increment
// method. The increment method takes an optional
// parameter. If the argument is not a number, then 1
// is used as the default.

var myObject = {
```

```
    value: 0,
```



```
document.writeln(myObject.value);    // 1

myObject.increment(2);
document.writeln(myObject.value);    // 3
```

A method can use `this` to access the object so that it can retrieve values from the object or modify the object. The binding of `this` to the object happens at invocation time. This very late binding makes functions that use `this` highly reusable. Methods that get their object context from `this` are called *public methods*.

The Function Invocation Pattern

When a function is not the property of an object, then it is invoked as a function:

```
var sum = add(3, 4);    // sum is 7
```

When a function is invoked with this pattern, `this` is bound to the global object. This was a mistake in the design of the language. Had the language been designed correctly, when the inner function is invoked, `this` would still be bound to the `this` variable of the outer function. A consequence of this error is that a method cannot employ an inner function to help it do its work because the inner function does not share the method's access to the object as its `this` is bound to the wrong value. Fortunately, there is an easy workaround. If the method defines a variable and assigns it the value of `this`, the inner function will have access to `this` through that variable. By convention, the name of that variable is `that`:

```
// Augment myObject with a double method.

myObject.double = function (  ) {
    var that = this;    // Workaround.

    var helper = function (  ) {
        that.value = add(that.value, that.value);


    helper(  );    // Invoke helper as a function.
};

// Invoke double as a method.
```

```
myObject.double(  );
```


JavaScript is a *prototypal* inheritance language. That means that objects can inherit properties directly from other objects. The language is class-free.

This is a radical departure from the current fashion. Most languages today are *classical*. Prototypal inheritance is powerfully expressive, but is not widely understood. JavaScript itself is not confident in its prototypal nature, so it offers an object-making syntax that is reminiscent of the classical languages. Few classical programmers found prototypal inheritance to be acceptable, and classically inspired syntax obscures the language's true prototypal nature. It is the worst of both worlds.

If a function is invoked with the `new` prefix, then a new object will be created with a hidden link to the value of the function's `prototype` member, and `this` will be bound to that new object.

The `new` prefix also changes the behavior of the `return` statement. We will see more about that next.

```
// Create a constructor function called Quo.
// It makes an object with a status property.

var Quo = function (string) {
    this.status = string;
};

// Give all instances of Quo a public method
// called get_status.

Quo.prototype.get_status = function (  ) {
    return this.status;
};

// Make an instance of Quo.

var myQuo = new Quo("confused");

document.writeln(myQuo.get_status(  ));  // confused
```

Functions that are intended to be used with the `new` prefix are called *constructors*. By convention, they are kept in variables with a capitalized name. If a constructor is called without the `new` prefix, very bad things can happen without a compile-time or runtime warning, so the capitalization convention is really important.

Use of this style of constructor functions is not recommended. We will see better alter-

Because JavaScript is a functional object-oriented language, functions can have meth-
ods.

The `apply` method lets us construct an array of arguments to use to invoke a function. It also lets us choose the value of `this`. The `apply` method takes two parameters. The first is the value that should be bound to `this`. The second is an array of parameters.

```
// Make an array of 2 numbers and add them.

var array = [3, 4];
var sum = add.apply(null, array);    // sum is 7

// Make an object with a status member.

var statusObject = {
    status: 'A-OK'
};

// statusObject does not inherit from Quo.prototype,
// but we can invoke the get_status method on
// statusObject even though statusObject does not have
// a get_status method.

var status = Quo.prototype.get_status.apply(statusObject);
    // status is 'A-OK'
```

JavaScript: The Good Parts by Douglas Crockford

JavaScript: The Good Parts by Douglas Crockford