Functions

Functions are an important part of any programming language, and prior to ECMAScript 6, JavaScript functions hadn't changed much since the language was created. This left a backlog of problems and nuanced behavior that made making mistakes easy and often required more code just to achieve very basic behaviors.

ECMAScript 6 functions make a big leap forward, taking into account years of complaints and requests from JavaScript developers. The result is a number of incremental improvements on top of ECMAScript 5 functions that make programming in JavaScript less error-prone and more powerful.

Functions with Default Parameter Values

Functions in JavaScript are unique in that they allow any number of parameters to be passed, regardless of the number of parameters declared in the function definition. This allows you to define functions that can handle different numbers of parameters, often by just filling in default values when parameters aren't provided. This section covers how default parameters work both in and prior to ECMAScript 6, along with some important information on the `arguments` object, using expressions as parameters, and another TDZ.

Simulating Default Parameter Values in ECMAScript 5

In ECMAScript 5 and earlier, you would likely use the following pattern to create a function with default parameters values:

```
function makeRequest(url, timeout, callback) {

    timeout = timeout || 2000;
    callback = callback || function() {};

    // the rest of the function

}
```

In this example, both `timeout` and `callback` are actually optional because they are given a default value if a parameter isn't provided. The logical OR operator (`||`) always returns the second operand when the first is falsy. Since named function parameters that are not explicitly provided are set to `undefined` , the logical OR operator is frequently used to provide default values for missing

parameters. There is a flaw with this approach, however, in that a valid value for `timeout` might actually be `0`, but this would replace it with `2000` because `0` is falsy.

In that case, a safer alternative is to check the type of the argument using `typeof`, as in this example:

```
function makeRequest(url, timeout, callback) {

    timeout = (typeof timeout !== "undefined") ? timeout : 2000;
    callback = (typeof callback !== "undefined") ? callback : function() {};

    // the rest of the function

}
```

While this approach is safer, it still requires a lot of extra code for a very basic operation. Popular JavaScript libraries are filled with similar patterns, as this represents a common pattern.

Default Parameter Values in ECMAScript 6

ECMAScript 6 makes it easier to provide default values for parameters by providing initializations that are used when the parameter isn't formally passed. For example:

```
function makeRequest(url, timeout = 2000, callback = function() {}) {

    // the rest of the function

}
```

This function only expects the first parameter to always be passed. The other two parameters have default values, which makes the body of the function much smaller because you don't need to add any code to check for a missing value.

When `makeRequest()` is called with all three parameters, the defaults are not used. For example:

```
// uses default timeout and callback
makeRequest("/foo");

// uses default callback
makeRequest("/foo", 500);

// doesn't use defaults
makeRequest("/foo", 500, function(body) {
    doSomething(body);
});
```

ECMAScript 6 considers `url` to be required, which is why `"/foo"` is passed in all three calls to `makeRequest()`. The two parameters with a default value are considered optional.

It's possible to specify default values for any arguments, including those that appear before arguments without default values in the function declaration. For example, this is fine:

```
function makeRequest(url, timeout = 2000, callback) {

    // the rest of the function

}
```

In this case, the default value for `timeout` will only be used if there is no second argument passed in or if the second argument is explicitly passed in as `undefined`, as in this example:

```
// uses default timeout
makeRequest("/foo", undefined, function(body) {
    doSomething(body);
});

// uses default timeout
makeRequest("/foo");

// doesn't use default timeout
makeRequest("/foo", null, function(body) {
    doSomething(body);
});
```

In the case of default parameter values, a value of `null` is considered to be valid, meaning that in the third call to `makeRequest()`, the default value for `timeout` will not be used.

How Default Parameter Values Affect the arguments Object

Just keep in mind that the behavior of the `arguments` object is different when default parameter values are present. In ECMAScript 5 nonstrict mode, the `arguments` object reflects changes in the named parameters of a function. Here's some code that illustrates how this works:

```
function mixArgs(first, second) {
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d";
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}

mixArgs("a", "b");
```

This outputs:

```
true
true
true
true
```

The `arguments` object is always updated in nonstrict mode to reflect changes in the named parameters. Thus, when `first` and `second` are assigned new values, `arguments[0]` and `arguments[1]` are updated accordingly, making all of the `===` comparisons resolve to `true`.

ECMAScript 5's strict mode, however, eliminates this confusing aspect of the `arguments` object. In strict mode, the `arguments` object does not reflect changes to the named parameters. Here's the `mixArgs()` function again, but in strict mode:

```
function mixArgs(first, second) {
    "use strict";

    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d"
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}

mixArgs("a", "b");
```

The call to `mixArgs()` outputs:

```
true
true
false
false
```

This time, changing `first` and `second` doesn't affect `arguments`, so the output behaves as you'd normally expect it to.

The `arguments` object in a function using ECMAScript 6 default parameter values, however, will always behave in the same manner as ECMAScript 5 strict mode, regardless of whether the function is explicitly running in strict mode. The presence of default parameter values triggers the `arguments` object to remain detached from the named parameters. This is a subtle but important detail because of how the `arguments` object may be used. Consider the following:

```
// not in strict mode
function mixArgs(first, second = "b") {
    console.log(arguments.length);
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d"
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}

mixArgs("a");
```

This outputs:

```
1
true
false
false
false
```

In this example, `arguments.length` is 1 because only one argument was passed to `mixArgs()`. That also means `arguments[1]` is `undefined`, which is the expected behavior when only one argument is passed to a function. That means `first` is equal to `arguments[0]` as well. Changing `first` and `second` has no effect on `arguments`. This behavior occurs in both nonstrict and strict mode, so you can rely on `arguments` to always reflect the initial call state.

Default Parameter Expressions

Perhaps the most interesting feature of default parameter values is that the default value need not be a primitive value. You can, for example, execute a function to retrieve the default parameter value, like this:

```
function getValue() {
    return 5;
}

function add(first, second = getValue()) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 6
```

Here, if the last argument isn't provided, the function `getValue()` is called to retrieve the correct default value. Keep in mind that `getValue()` is only called when `add()` is called without a second parameter, not when the function declaration is first parsed. That means if `getValue()` were written differently, it could potentially return a different value. For instance:

```
let value = 5;

function getValue() {
    return value++;
}

function add(first, second = getValue()) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 6
console.log(add(1));        // 7
```

In this example, `value` begins as five and increments each time `getValue()` is called. The first call to `add(1)` returns 6, while the second call to `add(1)` returns 7 because `value` was incremented. Because the default value for `second` is only evaluated when the function is called, changes to that value can be made at any time.

W> Be careful when using function calls as default parameter values. If you forget the parentheses, such as `second = getValue` in the last example, you are passing a reference to the function rather than the result of the function call.

This behavior introduces another interesting capability. You can use a previous parameter as the default for a later parameter. Here's an example:

```
function add(first, second = first) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 2
```

In this code, the parameter `second` is given a default value of `first`, meaning that passing in just one argument leaves both arguments with the same value. So `add(1, 1)` returns 2 just as `add(1)` returns 2. Taking this a step further, you can pass `first` into a function to get the value for `second` as follows:

```
function getValue(value) {
    return value + 5;
}

function add(first, second = getValue(first)) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 7
```

This example sets `second` equal to the value returned by `getValue(first)`, so while `add(1, 1)` still returns 2, `add(1)` returns 7 (1 + 6).

The ability to reference parameters from default parameter assignments works only for previous arguments, so earlier arguments do not have access to later arguments. For example:

```
function add(first = second, second) {
    return first + second;
}

console.log(add(1, 1));         // 2
console.log(add(undefined, 1)); // throws error
```

The call to `add(undefined, 1)` throws an error because `second` is defined after `first` and is therefore unavailable as a default value. To understand why that happens, it's important to revisit temporal dead zones.

Default Parameter Value Temporal Dead Zone

Chapter 1 introduced the temporal dead zone (TDZ) as it relates to `let` and `const`, and default parameter values also have a TDZ where parameters cannot be accessed. Similar to a `let`

declaration, each parameter creates a new identifier binding that can't be referenced before initialization without throwing an error. Parameter initialization happens when the function is called, either by passing a value for the parameter or by using the default parameter value.

To explore the default parameter value TDZ, consider this example from "Default Parameter Expressions" again:

```
function getValue(value) {
    return value + 5;
}

function add(first, second = getValue(first)) {
    return first + second;
}

console.log(add(1, 1));     // 2
console.log(add(1));        // 7
```

The calls to `add(1, 1)` and `add(1)` effectively execute the following code to create the `first` and `second` parameter values:

```
// JavaScript representation of call to add(1, 1)
let first = 1;
let second = 1;

// JavaScript representation of call to add(1)
let first = 1;
let second = getValue(first);
```

When the function `add()` is first executed, the bindings `first` and `second` are added to a parameter-specific TDZ (similar to how `let` behaves). So while `second` can be initialized with the value of `first` because `first` is always initialized at that time, the reverse is not true. Now, consider this rewritten `add()` function:

```
function add(first = second, second) {
    return first + second;
}

console.log(add(1, 1));         // 2
console.log(add(undefined, 1)); // throws error
```

The calls to `add(1, 1)` and `add(undefined, 1)` in this example now map to this code behind the scenes:

```
// JavaScript representation of call to add(1, 1)
let first = 1;
let second = 1;

// JavaScript representation of call to add(undefined, 1)
let first = second;
let second = 1;
```

In this example, the call to `add(undefined, 1)` throws an error because `second` hasn't yet been initialized when `first` is initialized. At that point, `second` is in the TDZ and therefore any references to `second` throw an error. This mirrors the behavior of `let` bindings discussed in Chapter 1.

I> Function parameters have their own scope and their own TDZ that is separate from the function body scope. That means the default value of a parameter cannot access any variables declared inside the function body.

Working with Unnamed Parameters

So far, the examples in this chapter have only covered parameters that have been named in the function definition. However, JavaScript functions don't limit the number of parameters that can be passed to the number of named parameters defined. You can always pass fewer or more parameters than formally specified. Default parameter values make it clear when a function can accept fewer parameters, and ECMAScript 6 sought to make the problem of passing more parameters than defined better as well.

Unnamed Parameters in ECMAScript 5

Early on, JavaScript provided the `arguments` object as a way to inspect all function parameters that are passed without necessarily defining each parameter individually. While inspecting `arguments` works fine in most cases, this object can be a little cumbersome to work with. For example, examine this code, which inspects the `arguments` object:

```
function pick(object) {
    let result = Object.create(null);

    // start at the second parameter
    for (let i = 1, len = arguments.length; i < len; i++) {
        result[arguments[i]] = object[arguments[i]];
    }

    return result;
}

let book = {
    title: "Understanding ECMAScript 6",
    author: "Nicholas C. Zakas",
    year: 2015
};

let bookData = pick(book, "author", "year");

console.log(bookData.author);   // "Nicholas C. Zakas"
console.log(bookData.year);     // 2015
```

This function mimics the `pick()` method from the *Underscore.js* library, which returns a copy of a given object with some specified subset of the original object's properties. This example defines only one argument and expects the first argument to be the object from which to copy properties. Every other argument passed is the name of a property that should be copied on the result.

There are a couple of things to notice about this `pick()` function. First, it's not at all obvious that the function can handle more than one parameter. You could define several more parameters, but you would always fall short of indicating that this function can take any number of parameters. Second, because the first parameter is named and used directly, when you look for the properties to copy, you have to start in the `arguments` object at index 1 instead of index 0. Remembering to use the appropriate indices with `arguments` isn't necessarily difficult, but it's one more thing to keep track of.

ECMAScript 6 introduces rest parameters to help with these issues.

Rest Parameters

A *rest parameter* is indicated by three dots (`...`) preceding a named parameter. That named parameter becomes an `Array` containing the rest of the parameters passed to the function, which is where the name "rest" parameters originates. For example, `pick()` can be rewritten using rest parameters, like this:

```
function pick(object, ...keys) {
    let result = Object.create(null);

    for (let i = 0, len = keys.length; i < len; i++) {
        result[keys[i]] = object[keys[i]];
    }

    return result;
}
```

In this version of the function, `keys` is a rest parameter that contains all parameters passed after `object` (unlike `arguments`, which contains all parameters including the first one). That means you can iterate over `keys` from beginning to end without worry. As a bonus, you can tell by looking at the function that it is capable of handling any number of parameters.

I> Rest parameters do not affect a function's `length` property, which indicates the number of named parameters for the function. The value of `length` for `pick()` in this example is 1 because only `object` counts towards this value.

Rest Parameter Restrictions

There are two restrictions on rest parameters. The first restriction is that there can be only one rest parameter, and the rest parameter must be last. For example, this code won't work:

```
// Syntax error: Can't have a named parameter after rest parameters
function pick(object, ...keys, last) {
    let result = Object.create(null);

    for (let i = 0, len = keys.length; i < len; i++) {
        result[keys[i]] = object[keys[i]];
    }

    return result;
}
```

Here, the parameter `last` follows the rest parameter `keys`, which would cause a syntax error.

The second restriction is that rest parameters cannot be used in an object literal setter. That means this code would also cause a syntax error:

```
let object = {

    // Syntax error: Can't use rest param in setter
    set name(...value) {
        // do something
    }
};
```

This restriction exists because object literal setters are restricted to a single argument. Rest parameters are, by definition, an infinite number of arguments, so they're not allowed in this context.

How Rest Parameters Affect the arguments Object

Rest parameters were designed to replace `arguments` in ECMAScript. Originally, ECMAScript 4 did away with `arguments` and added rest parameters to allow an unlimited number of arguments to be passed to functions. ECMAScript 4 never came into being, but this idea was kept around and reintroduced in ECMAScript 6, despite `arguments` not being removed from the language.

The `arguments` object works together with rest parameters by reflecting the arguments that were passed to the function when called, as illustrated in this program:

```
function checkArgs(...args) {
    console.log(args.length);
    console.log(arguments.length);
    console.log(args[0], arguments[0]);
    console.log(args[1], arguments[1]);
}

checkArgs("a", "b");
```

The call to `checkArgs()` outputs:

```
2
2
a a
b b
```

The `arguments` object always correctly reflects the parameters that were passed into a function regardless of rest parameter usage.

That's all you really need to know about rest parameters to get started using them.

Increased Capabilities of the Function Constructor

The `Function` constructor is an infrequently used part of JavaScript that allows you to dynamically create a new function. The arguments to the constructor are the parameters for the function and the function body, all as strings. Here's an example:

```
var add = new Function("first", "second", "return first + second");

console.log(add(1, 1));     // 2
```

ECMAScript 6 augments the capabilities of the `Function` constructor to allow default parameters and rest parameters. You need only add an equals sign and a value to the parameter names, as follows:

```
var add = new Function("first", "second = first",
        "return first + second");

console.log(add(1, 1));     // 2
console.log(add(1));        // 2
```

In this example, the parameter `second` is assigned the value of `first` when only one parameter is passed. The syntax is the same as for function declarations that don't use `Function`.

For rest parameters, just add the `...` before the last parameter, like this:

```
var pickFirst = new Function("...args", "return args[0]");

console.log(pickFirst(1, 2));   // 1
```

This code creates a function that uses only a single rest parameter and returns the first argument that was passed in.

The addition of default and rest parameters ensures that `Function` has all of the same capabilities as the declarative form of creating functions.

The Spread Operator

Closely related to rest parameters is the spread operator. While rest parameters allow you to specify that multiple independent arguments should be combined into an array, the spread operator allows you to specify an array that should be split and have its items passed in as separate arguments to a function. Consider the `Math.max()` method, which accepts any number of arguments and returns the one with the highest value. Here's a simple use case for this method:

```
let value1 = 25,
    value2 = 50;

console.log(Math.max(value1, value2));      // 50
```

When you're dealing with just two values, as in this example, `Math.max()` is very easy to use. The two values are passed in, and the higher value is returned. But what if you've been tracking values in an array, and now you want to find the highest value? The `Math.max()` method doesn't allow you to pass in an array, so in ECMAScript 5 and earlier, you'd be stuck either searching the array yourself or using `apply()` as follows:

```
let values = [25, 50, 75, 100]

console.log(Math.max.apply(Math, values));  // 100
```

This solution works, but using `apply()` in this manner is a bit confusing. It actually seems to obfuscate the true meaning of the code with additional syntax.

The ECMAScript 6 spread operator makes this case very simple. Instead of calling `apply()`, you can pass the array to `Math.max()` directly and prefix it with the same `...` pattern used with rest parameters. The JavaScript engine then splits the array into individual arguments and passes them in, like this:

```
let values = [25, 50, 75, 100]

// equivalent to
// console.log(Math.max(25, 50, 75, 100));
console.log(Math.max(...values));           // 100
```

Now the call to `Math.max()` looks a bit more conventional and avoids the complexity of specifying a `this`-binding (the first argument to `Math.max.apply()` in the previous example) for a simple mathematical operation.

You can mix and match the spread operator with other arguments as well. Suppose you want the smallest number returned from `Math.max()` to be 0 (just in case negative numbers sneak into the array). You can pass that argument separately and still use the spread operator for the other arguments, as follows:

```
let values = [-25, -50, -75, -100]

console.log(Math.max(...values, 0));        // 0
```

In this example, the last argument passed to `Math.max()` is `0`, which comes after the other arguments are passed in using the spread operator.

The spread operator for argument passing makes using arrays for function arguments much easier. You'll likely find it to be a suitable replacement for the `apply()` method in most circumstances.

In addition to the uses you've seen for default and rest parameters so far, in ECMAScript 6, you can also apply both parameter types to JavaScript's `Function` constructor.

ECMAScript 6's name Property

Identifying functions can be challenging in JavaScript given the various ways a function can be defined. Additionally, the prevalence of anonymous function expressions makes debugging a bit more difficult, often resulting in stack traces that are hard to read and decipher. For these reasons, ECMAScript 6 adds the `name` property to all functions.

Choosing Appropriate Names

All functions in an ECMAScript 6 program will have an appropriate value for their `name` property. To see this in action, look at the following example, which shows a function and function expression, and prints the `name` properties for both:

```
function doSomething() {
    // ...
}

var doAnotherThing = function() {
    // ...
};

console.log(doSomething.name);          // "doSomething"
console.log(doAnotherThing.name);       // "doAnotherThing"
```

In this code, `doSomething()` has a `name` property equal to `"doSomething"` because it's a function declaration. The anonymous function expression `doAnotherThing()` has a `name` of `"doAnotherThing"` because that's the name of the variable to which it is assigned.

Special Cases of the name Property

While appropriate names for function declarations and function expressions are easy to find, ECMAScript 6 goes further to ensure that *all* functions have appropriate names. To illustrate this, consider the following program:

```
var doSomething = function doSomethingElse() {
    // ...
};

var person = {
    get firstName() {
        return "Nicholas"
    },
    sayName: function() {
        console.log(this.name);
    }
}

console.log(doSomething.name);      // "doSomethingElse"
console.log(person.sayName.name);   // "sayName"

var descriptor = Object.getOwnPropertyDescriptor(person, "firstName");
console.log(descriptor.get.name); // "get firstName"
```

In this example, `doSomething.name` is `"doSomethingElse"` because the function expression itself has a name, and that name takes priority over the variable to which the function was assigned. The `name` property of `person.sayName()` is `"sayName"`, as the value was interpreted from the object literal. Similarly, `person.firstName` is actually a getter function, so its name is `"get firstName"` to indicate this difference. Setter functions are prefixed with `"set"` as well. (Both getter and setter functions must be retrieved using `Object.getOwnPropertyDescriptor()`.)

There are a couple of other special cases for function names, too. Functions created using `bind()` will have their names prefixed with `"bound"` and functions created using the `Function` constructor have a name of `"anonymous"`, as in this example:

```
var doSomething = function() {
    // ...
};

console.log(doSomething.bind().name);   // "bound doSomething"

console.log((new Function()).name);     // "anonymous"
```

The `name` of a bound function will always be the `name` of the function being bound prefixed with the string `"bound "`, so the bound version of `doSomething()` is `"bound doSomething"`.

Keep in mind that the value of `name` for any function does not necessarily refer to a variable of the same name. The `name` property is meant to be informative, to help with debugging, so there's no way to use the value of `name` to get a reference to the function.

Clarifying the Dual Purpose of Functions

In ECMAScript 5 and earlier, functions serve the dual purpose of being callable with or without `new`. When used with `new`, the `this` value inside a function is a new object and that new object is returned, as illustrated in this example:

```
function Person(name) {
    this.name = name;
}

var person = new Person("Nicholas");
var notAPerson = Person("Nicholas");

console.log(person);        // "[Object object]"
console.log(notAPerson);    // "undefined"
```

When creating `notAPerson`, calling `Person()` without `new` results in `undefined` (and sets a `name` property on the global object in nonstrict mode). The capitalization of `Person` is the only real indicator that the function is meant to be called using `new`, as is common in JavaScript programs. This confusion over the dual roles of functions led to some changes in ECMAScript 6.

JavaScript has two different internal-only methods for functions: `[[Call]]` and `[[Construct]]`. When a function is called without `new`, the `[[Call]]` method is executed, which executes the body of the function as it appears in the code. When a function is called with `new`, that's when the `[[Construct]]` method is called. The `[[Construct]]` method is responsible for creating a new object, called the *new target*, and then executing the function body with `this` set to the new target. Functions that have a `[[Construct]]` method are called *constructors*.

I> Keep in mind that not all functions have `[[Construct]]`, and therefore not all functions can be called with `new`. Arrow functions, discussed in the "Arrow Functions" section, do not have a `[[Construct]]` method.

Determining How a Function was Called in ECMAScript 5

The most popular way to determine if a function was called with `new` (and hence, with constructor) in ECMAScript 5 is to use `instanceof`, for example:

```
function Person(name) {
    if (this instanceof Person) {
        this.name = name;   // using new
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person("Nicholas");  // throws error
```

Here, the `this` value is checked to see if it's an instance of the constructor, and if so, execution continues as normal. If `this` isn't an instance of `Person`, then an error is thrown. This works because the `[[Construct]]` method creates a new instance of `Person` and assigns it to `this`. Unfortunately, this approach is not completely reliable because `this` can be an instance of `Person` without using `new`, as in this example:

```
function Person(name) {
    if (this instanceof Person) {
        this.name = name;   // using new
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person.call(person, "Michael");    // works!
```

The call to `Person.call()` passes the `person` variable as the first argument, which means `this` is set to `person` inside of the `Person` function. To the function, there's no way to distinguish this from being called with `new`.

The new.target MetaProperty

To solve this problem, ECMAScript 6 introduces the `new.target` *metaproperty*. A metaproperty is a property of a non-object that provides additional information related to its target (such as `new`). When a function's `[[Construct]]` method is called, `new.target` is filled with the target of the `new` operator. That target is typically the constructor of the newly created object instance that will become `this` inside the function body. If `[[Call]]` is executed, then `new.target` is `undefined`.

This new metaproperty allows you to safely detect if a function is called with `new` by checking whether `new.target` is defined as follows:

```
function Person(name) {
    if (typeof new.target !== "undefined") {
        this.name = name;   // using new
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person.call(person, "Michael");    // error!
```

By using `new.target` instead of `this instanceof Person`, the `Person` constructor is now correctly throwing an error when used without `new`.

You can also check that `new.target` was called with a specific constructor. For instance, look at this example:

```
function Person(name) {
    if (new.target === Person) {
        this.name = name;   // using new
    } else {
        throw new Error("You must use new with Person.")
    }
}

function AnotherPerson(name) {
    Person.call(this, name);
}

var person = new Person("Nicholas");
var anotherPerson = new AnotherPerson("Nicholas");  // error!
```

In this code, `new.target` must be `Person` in order to work correctly. When `new AnotherPerson("Nicholas")` is called, the subsequent call to `Person.call(this, name)` will throw an error because `new.target` is `undefined` inside of the `Person` constructor (it was called without `new`).

W> Warning: Using `new.target` outside of a function is a syntax error.

By adding `new.target`, ECMAScript 6 helped to clarify some ambiguity around functions calls. Following on this theme, ECMAScript 6 also addresses another previously ambiguous part of the language: declaring functions inside of blocks.

Block-Level Functions

In ECMAScript 3 and earlier, a function declaration occurring inside of a block (a *block-level function*) was technically a syntax error, but all browsers still supported it. Unfortunately, each browser that allowed the syntax behaved in a slightly different way, so it is considered a best practice to avoid function declarations inside of blocks (the best alternative is to use a function expression).

In an attempt to rein in this incompatible behavior, ECMAScript 5 strict mode introduced an error whenever a function declaration was used inside of a block in this way:

```
"use strict";

if (true) {

    // Throws a syntax error in ES5, not so in ES6
    function doSomething() {
        // ...
    }
}
```

In ECMAScript 5, this code throws a syntax error. In ECMAScript 6, the `doSomething()` function is considered a block-level declaration and can be accessed and called within the same block in which it was defined. For example:

```
"use strict";

if (true) {

    console.log(typeof doSomething);        // "function"

    function doSomething() {
        // ...
    }

    doSomething();
}

console.log(typeof doSomething);            // "undefined"
```

Block level functions are hoisted to the top of the block in which they are defined, so `typeof doSomething` returns `"function"` even though it appears before the function declaration in the code. Once the `if` block is finished executing, `doSomething()` no longer exists.

Deciding When to Use Block-Level Functions

Block level functions are similar to `let` function expressions in that the function definition is removed once execution flows out of the block in which it's defined. The key difference is that block

level functions are hoisted to the top of the containing block. Function expressions that use `let` are not hoisted, as this example illustrates:

```
"use strict";

if (true) {

    console.log(typeof doSomething);        // throws error

    let doSomething = function () {
        // ...
    }

    doSomething();
}

console.log(typeof doSomething);
```

Here, code execution stops when `typeof doSomething` is executed, because the `let` statement hasn't been executed yet, leaving `doSomething()` in the TDZ. Knowing this difference, you can choose whether to use block level functions or `let` expressions based on whether or not you want the hoisting behavior.

Block-Level Functions in Nonstrict Mode

ECMAScript 6 also allows block-level functions in nonstrict mode, but the behavior is slightly different. Instead of hoisting these declarations to the top of the block, they are hoisted all the way to the containing function or global environment. For example:

```
// ECMAScript 6 behavior
if (true) {

    console.log(typeof doSomething);        // "function"

    function doSomething() {
        // ...
    }

    doSomething();
}

console.log(typeof doSomething);            // "function"
```

In this example, `doSomething()` is hoisted into the global scope so that it still exists outside of the `if` block. ECMAScript 6 standardized this behavior to remove the incompatible browser behaviors

that previously existed, so all ECMAScript 6 runtimes should behave in the same way.

Allowing block-level functions improves your ability to declare functions in JavaScript, but ECMAScript 6 also introduced a completely new way to declare functions.

Arrow Functions

One of the most interesting new parts of ECMAScript 6 is the *arrow function*. Arrow functions are, as the name suggests, functions defined with a new syntax that uses an "arrow" (=>). But arrow functions behave differently than traditional JavaScript functions in a number of important ways:

- **No `this`, `super`, `arguments`, and `new.target` bindings** - The value of `this`, `super`, `arguments`, and `new.target` inside of the function is by the closest containing nonarrow function. (`super` is covered in Chapter 4.)
- **Cannot be called with `new`** - Arrow functions do not have a `[[Construct]]` method and therefore cannot be used as constructors. Arrow functions throw an error when used with `new`.
- **No prototype** - since you can't use `new` on an arrow function, there's no need for a prototype. The `prototype` property of an arrow function doesn't exist.
- **Can't change `this`** - The value of `this` inside of the function can't be changed. It remains the same throughout the entire lifecycle of the function.
- **No `arguments` object** - Since arrow functions have no `arguments` binding, you must rely on named and rest parameters to access function arguments.
- **No duplicate named parameters** - arrow functions cannot have duplicate named parameters in strict or nonstrict mode, as opposed to nonarrow functions that cannot have duplicate named parameters only in strict mode.

There are a few reasons for these differences. First and foremost, `this` binding is a common source of error in JavaScript. It's very easy to lose track of the `this` value inside a function, which can result in unintended program behavior, and arrow functions eliminate this confusion. Second, by limiting arrow functions to simply executing code with a single `this` value, JavaScript engines can more easily optimize these operations, unlike regular functions, which might be used as a constructor or otherwise modified.

The rest of the differences are also focused on reducing errors and ambiguities inside of arrow functions. By doing so, JavaScript engines are better able to optimize arrow function execution.

I> Note: Arrow functions also have a `name` property that follows the same rule as other functions.

Arrow Function Syntax

The syntax for arrow functions comes in many flavors depending upon what you're trying to accomplish. All variations begin with function arguments, followed by the arrow, followed by the body of the function. Both the arguments and the body can take different forms depending on usage. For example, the following arrow function takes a single argument and simply returns it:

```
var reflect = value => value;

// effectively equivalent to:

var reflect = function(value) {
    return value;
};
```

When there is only one argument for an arrow function, that one argument can be used directly without any further syntax. The arrow comes next and the expression to the right of the arrow is evaluated and returned. Even though there is no explicit `return` statement, this arrow function will return the first argument that is passed in.

If you are passing in more than one argument, then you must include parentheses around those arguments, like this:

```
var sum = (num1, num2) => num1 + num2;

// effectively equivalent to:

var sum = function(num1, num2) {
    return num1 + num2;
};
```

The `sum()` function simply adds two arguments together and returns the result. The only difference between this arrow function and the `reflect()` function is that the arguments are enclosed in parentheses with a comma separating them (like traditional functions).

If there are no arguments to the function, then you must include an empty set of parentheses in the declaration, as follows:

```
var getName = () => "Nicholas";

// effectively equivalent to:

var getName = function() {
    return "Nicholas";
};
```

When you want to provide a more traditional function body, perhaps consisting of more than one expression, then you need to wrap the function body in braces and explicitly define a return value, as in this version of `sum()`:

```
var sum = (num1, num2) => {
    return num1 + num2;
};

// effectively equivalent to:

var sum = function(num1, num2) {
    return num1 + num2;
};
```

You can more or less treat the inside of the curly braces the same as you would in a traditional function, with the exception that `arguments` is not available.

If you want to create a function that does nothing, then you need to include curly braces, like this:

```
var doNothing = () => {};

// effectively equivalent to:

var doNothing = function() {};
```

Curly braces are used to denote the function's body, which works just fine in the cases you've seen so far. But an arrow function that wants to return an object literal outside of a function body must wrap the literal in parentheses. For example:

```
var getTempItem = id => ({ id: id, name: "Temp" });

// effectively equivalent to:

var getTempItem = function(id) {

    return {
        id: id,
        name: "Temp"
    };
};
```

Wrapping the object literal in parentheses signals that the braces are an object literal instead of the function body.

Creating Immediately-Invoked Function Expressions

One popular use of functions in JavaScript is creating immediately-invoked function expressions (IIFEs). IIFEs allow you to define an anonymous function and call it immediately without saving a reference. This pattern comes in handy when you want to create a scope that is shielded from the rest of a program. For example:

```
let person = function(name) {

    return {
        getName: function() {
            return name;
        }
    };

}("Nicholas");

console.log(person.getName());      // "Nicholas"
```

In this code, the IIFE is used to create an object with a `getName()` method. The method uses the `name` argument as the return value, effectively making `name` a private member of the returned object.

You can accomplish the same thing using arrow functions, so long as you wrap the arrow function in parentheses:

```
let person = ((name) => {

    return {
        getName: function() {
            return name;
        }
    };

})("Nicholas");

console.log(person.getName());      // "Nicholas"
```

Note that the parentheses are only around the arrow function definition, and not around `("Nicholas")`. This is different from a formal function, where the parentheses can be placed outside of the passed-in parameters as well as just around the function definition.

No this Binding

One of the most common areas of error in JavaScript is the binding of `this` inside of functions. Since the value of `this` can change inside a single function depending on the context in which the

function is called, it's possible to mistakenly affect one object when you meant to affect another. Consider the following example:

```
var PageHandler = {

    id: "123456",

    init: function() {
        document.addEventListener("click", function(event) {
            this.doSomething(event.type);     // error
        }, false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

In this code, the object `PageHandler` is designed to handle interactions on the page. The `init()` method is called to set up the interactions, and that method in turn assigns an event handler to call `this.doSomething()` . However, this code doesn't work exactly as intended.

The call to `this.doSomething()` is broken because `this` is a reference to the object that was the target of the event (in this case `document`), instead of being bound to `PageHandler` . If you tried to run this code, you'd get an error when the event handler fires because `this.doSomething()` doesn't exist on the target `document` object.

You could fix this by binding the value of `this` to `PageHandler` explicitly using the `bind()` method on the function instead, like this:

```
var PageHandler = {

    id: "123456",

    init: function() {
        document.addEventListener("click", (function(event) {
            this.doSomething(event.type);     // no error
        }).bind(this), false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

Now the code works as expected, but it may look a little bit strange. By calling `bind(this)`, you're actually creating a new function whose `this` is bound to the current `this`, which is `PageHandler`. To avoid creating an extra function, a better way to fix this code is to use an arrow function.

Arrow functions have no `this` binding, which means the value of `this` inside an arrow function can only be determined by looking up the scope chain. If the arrow function is contained within a nonarrow function, `this` will be the same as the containing function; otherwise, `this` is equivalent to the value of `this` in the global scope. Here's one way you could write this code using an arrow function:

```
var PageHandler = {

    id: "123456",

    init: function() {
        document.addEventListener("click",
                event => this.doSomething(event.type), false);
    },

    doSomething: function(type) {
        console.log("Handling " + type  + " for " + this.id);
    }
};
```

The event handler in this example is an arrow function that calls `this.doSomething()`. The value of `this` is the same as it is within `init()`, so this version of the code works similarly to the one using `bind(this)`. Even though the `doSomething()` method doesn't return a value, it's still the only statement executed in the function body, and so there is no need to include braces.

Arrow functions are designed to be "throwaway" functions, and so cannot be used to define new types; this is evident from the missing `prototype` property, which regular functions have. If you try to use the `new` operator with an arrow function, you'll get an error, as in this example:

```
var MyType = () => {},
    object = new MyType();  // error - you can't use arrow functions with 'new'
```

In this code, the call to `new MyType()` fails because `MyType` is an arrow function and therefore has no `[[Construct]]` behavior. Knowing that arrow functions cannot be used with `new` allows JavaScript engines to further optimize their behavior.

Also, since the `this` value is determined by the containing function in which the arrow function is defined, you cannot change the value of `this` using `call()`, `apply()`, or `bind()`.

Arrow Functions and Arrays

The concise syntax for arrow functions makes them ideal for use with array processing, too. For example, if you want to sort an array using a custom comparator, you'd typically write something like this:

```
var result = values.sort(function(a, b) {
    return a - b;
});
```

That's a lot of syntax for a very simple procedure. Compare that to the more terse arrow function version:

```
var result = values.sort((a, b) => a - b);
```

The array methods that accept callback functions such as `sort()`, `map()`, and `reduce()` can all benefit from simpler arrow function syntax, which changes seemingly complex processes into simpler code.

No arguments Binding

Even though arrow functions don't have their own `arguments` object, it's possible for them to access the `arguments` object from a containing function. That `arguments` object is then available no matter where the arrow function is executed later on. For example:

```
function createArrowFunctionReturningFirstArg() {
    return () => arguments[0];
}

var arrowFunction = createArrowFunctionReturningFirstArg(5);

console.log(arrowFunction());       // 5
```

Inside `createArrowFunctionReturningFirstArg()`, the `arguments[0]` element is referenced by the created arrow function. That reference contains the first argument passed to the `createArrowFunctionReturningFirstArg()` function. When the arrow function is later executed, it returns `5`, which was the first argument passed to `createArrowFunctionReturningFirstArg()`. Even though the arrow function is no longer in the scope of the function that created it, `arguments` remains accessible due to scope chain resolution of the `arguments` identifier.

Identifying Arrow Functions

Despite the different syntax, arrow functions are still functions, and are identified as such. Consider the following code:

```
var comparator = (a, b) => a - b;

console.log(typeof comparator);                 // "function"
console.log(comparator instanceof Function);    // true
```

The `console.log()` output reveals that both `typeof` and `instanceof` behave the same with arrow functions as they do with other functions.

Also like other functions, you can still use `call()`, `apply()`, and `bind()` on arrow functions, although the `this`-binding of the function will not be affected. Here are some examples:

```
var sum = (num1, num2) => num1 + num2;

console.log(sum.call(null, 1, 2));      // 3
console.log(sum.apply(null, [1, 2]));   // 3

var boundSum = sum.bind(null, 1, 2);

console.log(boundSum());                // 3
```

The `sum()` function is called using `call()` and `apply()` to pass arguments, as you'd do with any function. The `bind()` method is used to create `boundSum()`, which has its two arguments bound to `1` and `2` so that they don't need to be passed directly.

Arrow functions are appropriate to use anywhere you're currently using an anonymous function expression, such as with callbacks. The next section covers another major ECMAScript 6 development, but this one is all internal, and has no new syntax.

Tail Call Optimization

Perhaps the most interesting change to functions in ECMAScript 6 is an engine optimization, which changes the tail call system. A *tail call* is when a function is called as the last statement in another function, like this:

```
function doSomething() {
    return doSomethingElse();   // tail call
}
```

Tail calls as implemented in ECMAScript 5 engines are handled just like any other function call: a new stack frame is created and pushed onto the call stack to represent the function call. That means every previous stack frame is kept in memory, which is problematic when the call stack gets too large.

What's Different?

ECMAScript 6 seeks to reduce the size of the call stack for certain tail calls in strict mode (nonstrict mode tail calls are left untouched). With this optimization, instead of creating a new stack frame for a tail call, the current stack frame is cleared and reused so long as the following conditions are met:

1. The tail call does not require access to variables in the current stack frame (meaning the function is not a closure)
2. The function making the tail call has no further work to do after the tail call returns
3. The result of the tail call is returned as the function value

As an example, this code can easily be optimized because it fits all three criteria:

```
"use strict";

function doSomething() {
    // optimized
    return doSomethingElse();
}
```

This function makes a tail call to `doSomethingElse()`, returns the result immediately, and doesn't access any variables in the local scope. One small change, not returning the result, results in an unoptimized function:

```
"use strict";

function doSomething() {
    // not optimized - no return
    doSomethingElse();
}
```

Similarly, if you have a function that performs an operation after returning from the tail call, then the function can't be optimized:

```
"use strict";

function doSomething() {
    // not optimized - must add after returning
    return 1 + doSomethingElse();
}
```

This example adds the result of `doSomethingElse()` with 1 before returning the value, and that's enough to turn off optimization.

Another common way to inadvertently turn off optimization is to store the result of a function call in a variable and then return the result, such as:

```
"use strict";

function doSomething() {
    // not optimized - call isn't in tail position
    var result = doSomethingElse();
    return result;
}
```

This example cannot be optimized because the value of `doSomethingElse()` isn't immediately returned.

Perhaps the hardest situation to avoid is in using closures. Because a closure has access to variables in the containing scope, tail call optimization may be turned off. For example:

```
"use strict";

function doSomething() {
    var num = 1,
        func = () => num;

    // not optimized - function is a closure
    return func();
}
```

The closure `func()` has access to the local variable `num` in this example. Even though the call to `func()` immediately returns the result, optimization can't occur due to referencing the variable `num`.

How to Harness Tail Call Optimization

In practice, tail call optimization happens behind-the-scenes, so you don't need to think about it unless you're trying to optimize a function. The primary use case for tail call optimization is in

recursive functions, as that is where the optimization has the greatest effect. Consider this function, which computes factorials:

```
function factorial(n) {

    if (n <= 1) {
        return 1;
    } else {

        // not optimized - must multiply after returning
        return n * factorial(n - 1);
    }
}
```

This version of the function cannot be optimized, because multiplication must happen after the recursive call to `factorial()`. If `n` is a very large number, the call stack size will grow and could potentially cause a stack overflow.

In order to optimize the function, you need to ensure that the multiplication doesn't happen after the last function call. To do this, you can use a default parameter to move the multiplication operation outside of the `return` statement. The resulting function carries along the temporary result into the next iteration, creating a function that behaves the same but *can* be optimized by an ECMAScript 6 engine. Here's the new code:

```
function factorial(n, p = 1) {

    if (n <= 1) {
        return 1 * p;
    } else {
        let result = n * p;

        // optimized
        return factorial(n - 1, result);
    }
}
```

In this rewritten version of `factorial()`, a second argument `p` is added as a parameter with a default value of 1. The `p` parameter holds the previous multiplication result so that the next result can be computed without another function call. When `n` is greater than 1, the multiplication is done first and then passed in as the second argument to `factorial()`. This allows the ECMAScript 6 engine to optimize the recursive call.

Tail call optimization is something you should think about whenever you're writing a recursive function, as it can provide a significant performance improvement, especially when applied in a

computationally-expensive function.

Summary

Functions haven't undergone a huge change in ECMAScript 6, but rather, a series of incremental changes that make them easier to work with.

Default function parameters allow you to easily specify what value to use when a particular argument isn't passed. Prior to ECMAScript 6, this would require some extra code inside the function, to both check for the presence of arguments and assign a different value.

Rest parameters allow you to specify an array into which all remaining parameters should be placed. Using a real array and letting you indicate which parameters to include makes rest parameters a much more flexible solution than `arguments`.

The spread operator is a companion to rest parameters, allowing you to deconstruct an array into separate parameters when calling a function. Prior to ECMAScript 6, there were only two ways to pass individual parameters contained in an array: by manually specifying each parameter or using `apply()`. With the spread operator, you can easily pass an array to any function without worrying about the `this` binding of the function.

The addition of the `name` property should help you more easily identify functions for debugging and evaluation purposes. Additionally, ECMAScript 6 formally defines the behavior of block-level functions so they are no longer a syntax error in strict mode.

In ECMAScript 6, the behavior of a function is defined by `[[Call]]`, normal function execution, and `[[Construct]]`, when a function is called with `new`. The `new.target` metaproperty also allows you to determine if a function was called using `new` or not.

The biggest change to functions in ECMAScript 6 was the addition of arrow functions. Arrow functions are designed to be used in place of anonymous function expressions. Arrow functions have a more concise syntax, lexical `this` binding, and no `arguments` object. Additionally, arrow functions can't change their `this` binding, and so can't be used as constructors.

Tail call optimization allows some function calls to be optimized in order to keep a smaller call stack, use less memory, and prevent stack overflow errors. This optimization is applied by the engine automatically when it is safe to do so, however, you may decide to rewrite recursive functions in order to take advantage of this optimization.