Chapter 6. Objects

Objects are JavaScript's most fundamental datatype, and you have already seen them many times in the chapters that precede this one. Because objects are so important to the JavaScript language, it is important that you understand how they work in detail, and this chapter provides that detail. It begins with a formal overview of objects, then dives into practical sections about creating objects and querying, setting, deleting, testing, and enumerating the properties of objects. These property-focused sections are followed by sections that explain how to extend, serialize, and define important methods on objects. Finally, the chapter concludes with a long section about new object literal syntax in ES6 and more recent versions of the language.

6.1 Introduction to Objects

An object is a composite value: it aggregates multiple values (primitive values or other objects) and allows you to store and retrieve those values by name. An object is an unordered collection of *properties*, each of which has a name and a value. Property names are usually strings (although, as we'll see in §6.10.3, property names can also be Symbols), so we can say that objects map strings to values. This string-to-value mapping goes by various names—you are probably already familiar with the fundamental data structure under the name "hash," "hashtable," "dictionary," or "associative array." An object is more than a simple string-to-value map, however. In addition to maintaining its own set of properties, a JavaScript object also inherits the properties of another object, known as its "prototype." The methods of an object are typically inherited properties, and this "prototypal inheritance" is a key feature of JavaScript.

JavaScript objects are dynamic—properties can usually be added and deleted—but they can be used to simulate the static objects and "structs" of statically typed languages. They can also be used (by ignoring the value part of the string-to-value mapping) to represent sets of strings.

Any value in JavaScript that is not a string, a number, a Symbol, or `true`, `false`, `null`, or `undefined` is an object. And even though strings, numbers, and booleans are not objects, they can behave like immutable objects.

Recall from §3.8 that objects are *mutable* and manipulated by reference rather than by value. If the variable `x` refers to an object and the code `let y = x;` is executed, the variable `y` holds a reference to the same object, not a copy of that object. Any modifications made to the object through the variable `y` are also visible through the variable `x`.

The most common things to do with objects are to create them and set, query, delete, test, and enumerate their properties. These fundamental operations are described in the opening sections of this chapter. The sections after that cover more advanced topics.

A *property* has a name and a value. A property name may be any string, including the empty string (or any Symbol), but no object may have two properties with the same name. The value may be any JavaScript value, or it may be a getter or setter function (or both). We'll learn about getter and setter functions in §6.10.6.

It is sometimes important to be able to distinguish between properties defined directly on an object and those that are inherited from a prototype object. JavaScript uses the term *own property* to refer to non-inherited properties.

In addition to its name and value, each property has three *property attributes*:

- The *writable* attribute specifies whether the value of the property can be set.
- The *enumerable* attribute specifies whether the property name is returned by a `for/in` loop.
- The *configurable* attribute specifies whether the property can be deleted and whether its attributes can be altered.

Many of JavaScript's built-in objects have properties that are read-only, non-enumerable, or non-configurable. By default, however, all properties of the objects you create are writable, enumerable, and configurable. §14.1 explains techniques for specifying non-default property attribute values for your objects.

6.2 Creating Objects

Objects can be created with object literals, with the `new` keyword, and with the `Object.create()` function. The subsections below describe each technique.

6.2.1 Object Literals

The easiest way to create an object is to include an object literal in your JavaScript code. In its simplest form, an *object literal* is a comma-separated list of colon-separated name:value pairs, enclosed within curly braces. A property name is a JavaScript identifier or a string literal (the empty string is allowed). A property value is any JavaScript expression; the value of the expression (it may be a primitive value or an object value) becomes the value of the property. Here are some examples:

```
let empty = {};                           // An object with no properties
let point = { x: 0, y: 0 };               // Two numeric properties
let p2 = { x: point.x, y: point.y+1 };    // More complex values
let book = {
    "main title": "JavaScript",           // These property names include spa
    "sub-title": "The Definitive Guide", // and hyphens, so use string liter
    for: "all audiences",                 // for is reserved, but no quotes.
    author: {                             // The value of this property is
        firstname: "David",               // itself an object.
        surname: "Flanagan"
    }
};
```

A trailing comma following the last property in an object literal is legal, and some programming styles encourage the use of these trailing commas so you're less likely to cause a syntax error if you add a new property at the end of the object literal at some later time.

An object literal is an expression that creates and initializes a new and distinct object each time it is evaluated. The value of each property is evaluated each time the literal is evaluated. This means that a single object literal can create many new objects if it appears within the body of a loop or in a function that is called repeatedly, and that the property values of these objects may differ from each other.

The object literals shown here use simple syntax that has been legal since the earliest versions of JavaScript. Recent versions of the language have introduced a number of new object literal features, which are covered in §6.10.

6.2.2 Creating Objects with new

The `new` operator creates and initializes a new object. The `new` keyword must be followed by a function invocation. A function used in this way is called a *constructor* and serves to initialize a newly created object. JavaScript includes constructors for its built-in types. For example:

```
let o = new Object();   // Create an empty object: same as {}.
let a = new Array();    // Create an empty array: same as [].
let d = new Date();     // Create a Date object representing the current time
let r = new Map();      // Create a Map object for key/value mapping
```

In addition to these built-in constructors, it is common to define your own constructor functions to initialize newly created objects. Doing so is covered in Chapter 9.

6.2.3 Prototypes

Before we can cover the third object creation technique, we must pause for a moment to explain prototypes. Almost every JavaScript object has a second JavaScript object associated with it. This second object is known as a *prototype*, and the first object inherits properties from the prototype.

All objects created by object literals have the same prototype object, and we can refer to this prototype object in JavaScript code as `Object.prototype`. Objects created using the `new` keyword and a constructor invocation use the value of the `prototype` property of the constructor function as their prototype. So the object created by `new Object()` inherits from `Object.prototype`, just as the object created by `{}` does. Similarly, the object created by `new Array()` uses `Array.prototype` as its prototype, and the object created by `new Date()` uses `Date.prototype` as its prototype. This can be confusing when first learning JavaScript. Remember: almost all objects have a *prototype*, but only a relatively small number of objects have a `prototype` property. It is these objects with `prototype` properties that define the *prototypes* for all the other objects.

`Object.prototype` is one of the rare objects that has no prototype: it does not inherit any properties. Other prototype objects are normal objects that do have a prototype. Most built-in constructors (and most user-defined constructors) have a prototype that inherits from

`Object.prototype`. For example, `Date.prototype` inherits properties from `Object.prototype`, so a Date object created by `new Date()` inherits properties from both `Date.prototype` and `Object.prototype`. This linked series of prototype objects is known as a *prototype chain*.

An explanation of how property inheritance works is in §6.3.2. Chapter 9 explains the connection between prototypes and constructors in more detail: it shows how to define new "classes" of objects by writing a constructor function and setting its `prototype` property to the prototype object to be used by the "instances" created with that constructor. And we'll learn how to query (and even change) the prototype of an object in §14.3.

6.2.4 Object.create()

`Object.create()` creates a new object, using its first argument as the prototype of that object:

```
let o1 = Object.create({x: 1, y: 2});      // o1 inherits properties x and y.
o1.x + o1.y                                // => 3
```

You can pass `null` to create a new object that does not have a prototype, but if you do this, the newly created object will not inherit anything, not even basic methods like `toString()` (which means it won't work with the `+` operator either):

```
let o2 = Object.create(null);              // o2 inherits no props or methods
```

If you want to create an ordinary empty object (like the object returned by `{}` or `new Object()`), pass `Object.prototype`:

```
let o3 = Object.create(Object.prototype); // o3 is like {} or new Object().
```

The ability to create a new object with an arbitrary prototype is a powerful one, and we'll use `Object.create()` in a number of places throughout this chapter. (`Object.create()` also takes an optional second argument that describes the properties of the new object. This second argument is an advanced feature covered in §14.1.)

One use for `Object.create()` is when you want to guard against unintended (but nonmalicious) modification of an object by a library function

that you don't have control over. Instead of passing the object directly to the function, you can pass an object that inherits from it. If the function reads properties of that object, it will see the inherited values. If it sets properties, however, those writes will not affect the original object.

```
let o = { x: "don't change this value" };
library.function(Object.create(o));  // Guard against accidental modificatio
```

To understand why this works, you need to know how properties are queried and set in JavaScript. These are the topics of the next section.

6.3 Querying and Setting Properties

To obtain the value of a property, use the dot (`.`) or square bracket (`[]`) operators described in §4.4. The lefthand side should be an expression whose value is an object. If using the dot operator, the righthand side must be a simple identifier that names the property. If using square brackets, the value within the brackets must be an expression that evaluates to a string that contains the desired property name:

```
let author = book.author;        // Get the "author" property of the book.
let name = author.surname;       // Get the "surname" property of the author.
let title = book["main title"]; // Get the "main title" property of the book
```

To create or set a property, use a dot or square brackets as you would to query the property, but put them on the lefthand side of an assignment expression:

```
book.edition = 7;                    // Create an "edition" property of book.
book["main title"] = "ECMAScript";  // Change the "main title" property.
```

When using square bracket notation, we've said that the expression inside the square brackets must evaluate to a string. A more precise statement is that the expression must evaluate to a string or a value that can be converted to a string or to a Symbol (§6.10.3). In Chapter 7, for example, we'll see that it is common to use numbers inside the square brackets.

6.3.1 Objects As Associative Arrays

As explained in the preceding section, the following two JavaScript expressions have the same value:

```
object.property
object["property"]
```

The first syntax, using the dot and an identifier, is like the syntax used to access a static field of a struct or object in C or Java. The second syntax, using square brackets and a string, looks like array access, but to an array indexed by strings rather than by numbers. This kind of array is known as an *associative array* (or hash or map or dictionary). JavaScript objects are associative arrays, and this section explains why that is important.

In C, C++, Java, and similar strongly typed languages, an object can have only a fixed number of properties, and the names of these properties must be defined in advance. Since JavaScript is a loosely typed language, this rule does not apply: a program can create any number of properties in any object. When you use the `.` operator to access a property of an object, however, the name of the property is expressed as an identifier. Identifiers must be typed literally into your JavaScript program; they are not a datatype, so they cannot be manipulated by the program.

On the other hand, when you access a property of an object with the `[]` array notation, the name of the property is expressed as a string. Strings are JavaScript datatypes, so they can be manipulated and created while a program is running. So, for example, you can write the following code in JavaScript:

```
let addr = "";
for(let i = 0; i < 4; i++) {
    addr += customer[`address${i}`] + "\n";
}
```

This code reads and concatenates the `address0`, `address1`, `address2`, and `address3` properties of the `customer` object.

This brief example demonstrates the flexibility of using array notation to access properties of an object with string expressions. This code could be rewritten using the dot notation, but there are cases in which only the array notation will do. Suppose, for example, that you are writing a program that uses network resources to compute the current value of the

user's stock market investments. The program allows the user to type in the name of each stock they own as well as the number of shares of each stock. You might use an object named `portfolio` to hold this information. The object has one property for each stock. The name of the property is the name of the stock, and the property value is the number of shares of that stock. So, for example, if a user holds 50 shares of stock in IBM, the `portfolio.ibm` property has the value `50`.

Part of this program might be a function for adding a new stock to the portfolio:

```
function addstock(portfolio, stockname, shares) {
    portfolio[stockname] = shares;
}
```

Since the user enters stock names at runtime, there is no way that you can know the property names ahead of time. Since you can't know the property names when you write the program, there is no way you can use the `.` operator to access the properties of the `portfolio` object. You can use the `[]` operator, however, because it uses a string value (which is dynamic and can change at runtime) rather than an identifier (which is static and must be hardcoded in the program) to name the property.

In Chapter 5, we introduced the `for/in` loop (and we'll see it again shortly, in §6.6). The power of this JavaScript statement becomes clear when you consider its use with associative arrays. Here is how you would use it when computing the total value of a portfolio:

```
function computeValue(portfolio) {
    let total = 0.0;
    for(let stock in portfolio) {        // For each stock in the portfolio:
        let shares = portfolio[stock];   // get the number of shares
        let price = getQuote(stock);     // look up share price
        total += shares * price;         // add stock value to total value
    }
    return total;                        // Return total value.
}
```

JavaScript objects are commonly used as associative arrays as shown here, and it is important to understand how this works. In ES6 and later, however, the Map class described in §11.1.2 is often a better choice than using a plain object.

6.3.2 Inheritance

JavaScript objects have a set of "own properties," and they also inherit a set of properties from their prototype object. To understand this, we must consider property access in more detail. The examples in this section use the `Object.create()` function to create objects with specified prototypes. We'll see in Chapter 9, however, that every time you create an instance of a class with `new`, you are creating an object that inherits properties from a prototype object.

Suppose you query the property `x` in the object `o`. If `o` does not have an own property with that name, the prototype object of `o`[1] is queried for the property `x`. If the prototype object does not have an own property by that name, but has a prototype itself, the query is performed on the prototype of the prototype. This continues until the property `x` is found or until an object with a `null` prototype is searched. As you can see, the `prototype` attribute of an object creates a chain or linked list from which properties are inherited:

```
let o = {};                 // o inherits object methods from Object.prototype
o.x = 1;                    // and it now has an own property x.
let p = Object.create(o); // p inherits properties from o and Object.prototyp
p.y = 2;                    // and has an own property y.
let q = Object.create(p); // q inherits properties from p, o, and...
q.z = 3;                    // ...Object.prototype and has an own property z.
let f = q.toString();       // toString is inherited from Object.prototype
q.x + q.y                   // => 3; x and y are inherited from o and p
```

Now suppose you assign to the property `x` of the object `o`. If `o` already has an own (non-inherited) property named `x`, then the assignment simply changes the value of this existing property. Otherwise, the assignment creates a new property named `x` on the object `o`. If `o` previously inherited the property `x`, that inherited property is now hidden by the newly created own property with the same name.

Property assignment examines the prototype chain only to determine whether the assignment is allowed. If `o` inherits a read-only property named `x`, for example, then the assignment is not allowed. (Details about when a property may be set are in §6.3.3.) If the assignment is allowed, however, it always creates or sets a property in the original object and never modifies objects in the prototype chain. The fact that inheritance occurs when querying properties but not when setting them is a key fea-

ture of JavaScript because it allows us to selectively override inherited properties:

```
let unitcircle = { r: 1 };           // An object to inherit from
let c = Object.create(unitcircle); // c inherits the property r
c.x = 1; c.y = 1;                    // c defines two properties of its own
c.r = 2;                             // c overrides its inherited property
unitcircle.r                         // => 1: the prototype is not affected
```

There is one exception to the rule that a property assignment either fails or creates or sets a property in the original object. If `o` inherits the property `x`, and that property is an accessor property with a setter method (see §6.10.6), then that setter method is called rather than creating a new property `x` in `o`. Note, however, that the setter method is called on the object `o`, not on the prototype object that defines the property, so if the setter method defines any properties, it will do so on `o`, and it will again leave the prototype chain unmodified.

6.3.3 Property Access Errors

Property access expressions do not always return or set a value. This section explains the things that can go wrong when you query or set a property.

It is not an error to query a property that does not exist. If the property `x` is not found as an own property or an inherited property of `o`, the property access expression `o.x` evaluates to `undefined`. Recall that our book object has a "sub-title" property, but not a "subtitle" property:

```
book.subtitle    // => undefined: property doesn't exist
```

It is an error, however, to attempt to query a property of an object that does not exist. The `null` and `undefined` values have no properties, and it is an error to query properties of these values. Continuing the preceding example:

```
let len = book.subtitle.length; // !TypeError: undefined doesn't have length
```

Property access expressions will fail if the lefthand side of the `.` is `null` or `undefined`. So when writing an expression like `book.author.surname`, you should be careful if you are not certain that

`book` and `book.author` are actually defined. Here are two ways to guard against this kind of problem:

```
// A verbose and explicit technique
let surname = undefined;
if (book) {
    if (book.author) {
        surname = book.author.surname;
    }
}

// A concise and idiomatic alternative to get surname or null or undefined
surname = book && book.author && book.author.surname;
```

To understand why this idiomatic expression works to prevent TypeError exceptions, you might want to review the short-circuiting behavior of the `&&` operator in §4.10.1.

As described in §4.4.1, ES2020 supports conditional property access with `?.`, which allows us to rewrite the previous assignment expression as:

```
let surname = book?.author?.surname;
```

Attempting to set a property on `null` or `undefined` also causes a TypeError. Attempts to set properties on other values do not always succeed, either: some properties are read-only and cannot be set, and some objects do not allow the addition of new properties. In strict mode (§5.6.3), a TypeError is thrown whenever an attempt to set a property fails. Outside of strict mode, these failures are usually silent.

The rules that specify when a property assignment succeeds and when it fails are intuitive but difficult to express concisely. An attempt to set a property `p` of an object `o` fails in these circumstances:

- `o` has an own property `p` that is read-only: it is not possible to set read-only properties.
- `o` has an inherited property `p` that is read-only: it is not possible to hide an inherited read-only property with an own property of the same name.
- `o` does not have an own property `p`; `o` does not inherit a property `p` with a setter method, and `o`'s *extensible* attribute (see §14.2) is `false`. Since `p` does not already exist in `o`, and if there is no setter

method to call, then `p` must be added to `o`. But if `o` is not extensible, then no new properties can be defined on it.

6.4 Deleting Properties

The `delete` operator (§4.13.4) removes a property from an object. Its single operand should be a property access expression. Surprisingly, `delete` does not operate on the value of the property but on the property itself:

```
delete book.author;           // The book object now has no author property.
delete book["main title"];    // Now it doesn't have "main title", either.
```

The `delete` operator only deletes own properties, not inherited ones. (To delete an inherited property, you must delete it from the prototype object in which it is defined. Doing this affects every object that inherits from that prototype.)

A `delete` expression evaluates to `true` if the delete succeeded or if the delete had no effect (such as deleting a nonexistent property). `delete` also evaluates to `true` when used (meaninglessly) with an expression that is not a property access expression:

```
let o = {x: 1};    // o has own property x and inherits property toString
delete o.x         // => true: deletes property x
delete o.x         // => true: does nothing (x doesn't exist) but true anywa
delete o.toString  // => true: does nothing (toString isn't an own property)
delete 1           // => true: nonsense, but true anyway
```

`delete` does not remove properties that have a *configurable* attribute of `false`. Certain properties of built-in objects are non-configurable, as are properties of the global object created by variable declaration and function declaration. In strict mode, attempting to delete a non-configurable property causes a TypeError. In non-strict mode, `delete` simply evaluates to `false` in this case:

```
// In strict mode, all these deletions throw TypeError instead of returning
delete Object.prototype // => false: property is non-configurable
var x = 1;              // Declare a global variable
delete globalThis.x     // => false: can't delete this property
```

```
function f() {}          // Declare a global function
delete globalThis.f      // => false: can't delete this property either
```

When deleting configurable properties of the global object in non-strict mode, you can omit the reference to the global object and simply follow the `delete` operator with the property name:

```
globalThis.x = 1;        // Create a configurable global property (no let or
delete x                 // => true: this property can be deleted
```

In strict mode, however, `delete` raises a SyntaxError if its operand is an unqualified identifier like `x`, and you have to be explicit about the property access:

```
delete x;                // SyntaxError in strict mode
delete globalThis.x;     // This works
```

6.5 Testing Properties

JavaScript objects can be thought of as sets of properties, and it is often useful to be able to test for membership in the set—to check whether an object has a property with a given name. You can do this with the `in` operator, with the `hasOwnProperty()` and `propertyIsEnumerable()` methods, or simply by querying the property. The examples shown here all use strings as property names, but they also work with Symbols (§6.10.3).

The `in` operator expects a property name on its left side and an object on its right. It returns `true` if the object has an own property or an inherited property by that name:

```
let o = { x: 1 };
"x" in o          // => true: o has an own property "x"
"y" in o          // => false: o doesn't have a property "y"
"toString" in o   // => true: o inherits a toString property
```

The `hasOwnProperty()` method of an object tests whether that object has an own property with the given name. It returns `false` for inherited properties:

```
let o = { x: 1 };
o.hasOwnProperty("x")         // => true: o has an own property x
o.hasOwnProperty("y")         // => false: o doesn't have a property y
o.hasOwnProperty("toString") // => false: toString is an inherited property
```

The `propertyIsEnumerable()` refines the `hasOwnProperty()` test. It returns `true` only if the named property is an own property and its *enumerable* attribute is `true`. Certain built-in properties are not enumerable. Properties created by normal JavaScript code are enumerable unless you've used one of the techniques shown in §14.1 to make them non-enumerable.

```
let o = { x: 1 };
o.propertyIsEnumerable("x")  // => true: o has an own enumerable property x
o.propertyIsEnumerable("toString")  // => false: not an own property
Object.prototype.propertyIsEnumerable("toString") // => false: not enumerable
```

Instead of using the `in` operator, it is often sufficient to simply query the property and use `!==` to make sure it is not undefined:

```
let o = { x: 1 };
o.x !== undefined        // => true: o has a property x
o.y !== undefined        // => false: o doesn't have a property y
o.toString !== undefined // => true: o inherits a toString property
```

There is one thing the `in` operator can do that the simple property access technique shown here cannot do. `in` can distinguish between properties that do not exist and properties that exist but have been set to `undefined`. Consider this code:

```
let o = { x: undefined };  // Property is explicitly set to undefined
o.x !== undefined          // => false: property exists but is undefined
o.y !== undefined          // => false: property doesn't even exist
"x" in o                   // => true: the property exists
"y" in o                   // => false: the property doesn't exist
delete o.x;                // Delete the property x
"x" in o                   // => false: it doesn't exist anymore
```

6.6 Enumerating Properties

Instead of testing for the existence of individual properties, we sometimes want to iterate through or obtain a list of all the properties of an object. There are a few different ways to do this.

The `for/in` loop was covered in §5.4.5. It runs the body of the loop once for each enumerable property (own or inherited) of the specified object, assigning the name of the property to the loop variable. Built-in methods that objects inherit are not enumerable, but the properties that your code adds to objects are enumerable by default. For example:

```
let o = {x: 1, y: 2, z: 3};           // Three enumerable own properties
o.propertyIsEnumerable("toString")    // => false: not enumerable
for(let p in o) {                     // Loop through the properties
    console.log(p);                   // Prints x, y, and z, but not toString
}
```

To guard against enumerating inherited properties with `for/in`, you can add an explicit check inside the loop body:

```
for(let p in o) {
    if (!o.hasOwnProperty(p)) continue;       // Skip inherited properties
}

for(let p in o) {
    if (typeof o[p] === "function") continue; // Skip all methods
}
```

As an alternative to using a `for/in` loop, it is often easier to get an array of property names for an object and then loop through that array with a `for/of` loop. There are four functions you can use to get an array of property names:

- `Object.keys()` returns an array of the names of the enumerable own properties of an object. It does not include non-enumerable properties, inherited properties, or properties whose name is a Symbol (see §6.10.3).
- `Object.getOwnPropertyNames()` works like `Object.keys()` but returns an array of the names of non-enumerable own properties as well, as long as their names are strings.
- `Object.getOwnPropertySymbols()` returns own properties whose names are Symbols, whether or not they are enumerable.

- `Reflect.ownKeys()` returns all own property names, both enumerable and non-enumerable, and both string and Symbol. (See §14.6.)

There are examples of the use of `Object.keys()` with a `for/of` loop in §6.7.

6.6.1 Property Enumeration Order

ES6 formally defines the order in which the own properties of an object are enumerated. `Object.keys()`, `Object.getOwnPropertyNames()`, `Object.getOwnPropertySymbols()`, `Reflect.ownKeys()`, and related methods such as `JSON.stringify()` all list properties in the following order, subject to their own additional constraints about whether they list non-enumerable properties or properties whose names are strings or Symbols:

- String properties whose names are non-negative integers are listed first, in numeric order from smallest to largest. This rule means that arrays and array-like objects will have their properties enumerated in order.
- After all properties that look like array indexes are listed, all remaining properties with string names are listed (including properties that look like negative numbers or floating-point numbers). These properties are listed in the order in which they were added to the object. For properties defined in an object literal, this order is the same order they appear in the literal.
- Finally, the properties whose names are Symbol objects are listed in the order in which they were added to the object.

The enumeration order for the `for/in` loop is not as tightly specified as it is for these enumeration functions, but implementations typically enumerate own properties in the order just described, then travel up the prototype chain enumerating properties in the same order for each prototype object. Note, however, that a property will not be enumerated if a property by that same name has already been enumerated, or even if a non-enumerable property by the same name has already been considered.

6.7 Extending Objects

A common operation in JavaScript programs is needing to copy the properties of one object to another object. It is easy to do that with code like this:

```
let target = {x: 1}, source = {y: 2, z: 3};
for(let key of Object.keys(source)) {
    target[key] = source[key];
}
target  // => {x: 1, y: 2, z: 3}
```

But because this is a common operation, various JavaScript frameworks have defined utility functions, often named `extend()`, to perform this copying operation. Finally, in ES6, this ability comes to the core JavaScript language in the form of `Object.assign()`.

`Object.assign()` expects two or more objects as its arguments. It modifies and returns the first argument, which is the target object, but does not alter the second or any subsequent arguments, which are the source objects. For each source object, it copies the enumerable own properties of that object (including those whose names are Symbols) into the target object. It processes the source objects in argument list order so that properties in the first source object override properties by the same name in the target object and properties in the second source object (if there is one) override properties with the same name in the first source object.

`Object.assign()` copies properties with ordinary property get and set operations, so if a source object has a getter method or the target object has a setter method, they will be invoked during the copy, but they will not themselves be copied.

One reason to assign properties from one object into another is when you have an object that defines default values for many properties and you want to copy those default properties into another object if a property by that name does not already exist in that object. Using `Object.assign()` naively will not do what you want:

```
Object.assign(o, defaults);  // overwrites everything in o with defaults
```

Instead, what you can do is to create a new object, copy the defaults into it, and then override those defaults with the properties in `o`:

```
o = Object.assign({}, defaults, o);
```

We'll see in §6.10.4 that you can also express this object copy-and-override operation using the `...` spread operator like this:

```
o = {...defaults, ...o};
```

We could also avoid the overhead of the extra object creation and copying by writing a version of `Object.assign()` that copies properties only if they are missing:

```
// Like Object.assign() but doesn't override existing properties
// (and also doesn't handle Symbol properties)
function merge(target, ...sources) {
    for(let source of sources) {
        for(let key of Object.keys(source)) {
            if (!(key in target)) { // This is different than Object.assign(
                target[key] = source[key];
            }
        }
    }
    return target;
}
Object.assign({x: 1}, {x: 2, y: 2}, {y: 3, z: 4})  // => {x: 2, y: 3, z: 4}
merge({x: 1}, {x: 2, y: 2}, {y: 3, z: 4})          // => {x: 1, y: 2, z: 4}
```

It is straightforward to write other property manipulation utilities like this `merge()` function. A `restrict()` function could delete properties of an object if they do not appear in another template object, for example. Or a `subtract()` function could remove all of the properties of one object from another object.

6.8 Serializing Objects

Object *serialization* is the process of converting an object's state to a string from which it can later be restored. The functions `JSON.stringify()` and `JSON.parse()` serialize and restore JavaScript objects. These functions use the JSON data interchange format. JSON stands for "JavaScript Object Notation," and its syntax is very similar to that of JavaScript object and array literals:

```
let o = {x: 1, y: {z: [false, null, ""]}}; // Define a test object
let s = JSON.stringify(o);    // s == '{"x":1,"y":{"z":[false,null,""]}}'
let p = JSON.parse(s);        // p == {x: 1, y: {z: [false, null, ""]}}
```

JSON syntax is a *subset* of JavaScript syntax, and it cannot represent all JavaScript values. Objects, arrays, strings, finite numbers, `true`, `false`, and `null` are supported and can be serialized and restored. `NaN`, `Infinity`, and `-Infinity` are serialized to `null`. Date objects are serialized to ISO-formatted date strings (see the `Date.toJSON()` function), but `JSON.parse()` leaves these in string form and does not restore the original Date object. Function, RegExp, and Error objects and the `undefined` value cannot be serialized or restored. `JSON.stringify()` serializes only the enumerable own properties of an object. If a property value cannot be serialized, that property is simply omitted from the stringified output. Both `JSON.stringify()` and `JSON.parse()` accept optional second arguments that can be used to customize the serialization and/or restoration process by specifying a list of properties to be serialized, for example, or by converting certain values during the serialization or stringification process. Complete documentation for these functions is in §11.6.

6.9 Object Methods

As discussed earlier, all JavaScript objects (except those explicitly created without a prototype) inherit properties from `Object.prototype`. These inherited properties are primarily methods, and because they are universally available, they are of particular interest to JavaScript programmers. We've already seen the `hasOwnProperty()` and `propertyIsEnumerable()` methods, for example. (And we've also already covered quite a few static functions defined on the `Object` constructor, such as `Object.create()` and `Object.keys()`.) This section explains a handful of universal object methods that are defined on `Object.prototype`, but which are intended to be replaced by other, more specialized implementations. In the sections that follow, we show examples of defining these methods on a single object. In Chapter 9, you'll learn how to define these methods more generally for an entire class of objects.

6.9.1 The toString() Method

The `toString()` method takes no arguments; it returns a string that somehow represents the value of the object on which it is invoked. JavaScript invokes this method of an object whenever it needs to convert the object to a string. This occurs, for example, when you use the `+` operator to concatenate a string with an object or when you pass an object to a method that expects a string.

The default `toString()` method is not very informative (though it is useful for determining the class of an object, as we will see in §14.4.3). For example, the following line of code simply evaluates to the string "[object Object]":

```
let s = { x: 1, y: 1 }.toString();  // s == "[object Object]"
```

Because this default method does not display much useful information, many classes define their own versions of `toString()`. For example, when an array is converted to a string, you obtain a list of the array elements, themselves each converted to a string, and when a function is converted to a string, you obtain the source code for the function. You might define your own `toString()` method like this:

```
let point = {
    x: 1,
    y: 2,
    toString: function() { return `(${this.x}, ${this.y})`; }
};
String(point)    // => "(1, 2)": toString() is used for string conversions
```

6.9.2 The toLocaleString() Method

In addition to the basic `toString()` method, objects all have a `toLocaleString()`. The purpose of this method is to return a localized string representation of the object. The default `toLocaleString()` method defined by Object doesn't do any localization itself: it simply calls `toString()` and returns that value. The Date and Number classes define customized versions of `toLocaleString()` that attempt to format numbers, dates, and times according to local conventions. Array defines a `toLocaleString()` method that works like `toString()` except that it formats array elements by calling their `toLocaleString()` methods instead of their `toString()` methods. You might do the same thing with a `point` object like this:

```
let point = {
    x: 1000,
    y: 2000,
    toString: function() { return `(${this.x}, ${this.y})`; },
    toLocaleString: function() {
        return `(${this.x.toLocaleString()}, ${this.y.toLocaleString()})`;
    }
};
point.toString()         // => "(1000, 2000)"
point.toLocaleString()   // => "(1,000, 2,000)": note thousands separators
```

The internationalization classes documented in §11.7 can be useful when implementing a `toLocaleString()` method.

6.9.3 The valueOf() Method

The `valueOf()` method is much like the `toString()` method, but it is called when JavaScript needs to convert an object to some primitive type other than a string—typically, a number. JavaScript calls this method automatically if an object is used in a context where a primitive value is required. The default `valueOf()` method does nothing interesting, but some of the built-in classes define their own `valueOf()` method. The Date class defines `valueOf()` to convert dates to numbers, and this allows Date objects to be chronologically compared with `<` and `>`. You could do something similar with a point object, defining a `valueOf()` method that returns the distance from the origin to the point:

```
let point = {
    x: 3,
    y: 4,
    valueOf: function() { return Math.hypot(this.x, this.y); }
};
Number(point)  // => 5: valueOf() is used for conversions to numbers
point > 4      // => true
point > 5      // => false
point < 6      // => true
```

6.9.4 The toJSON() Method

`Object.prototype` does not actually define a `toJSON()` method, but the `JSON.stringify()` method (see §6.8) looks for a `toJSON()` method on any object it is asked to serialize. If this method exists on the object to

be serialized, it is invoked, and the return value is serialized, instead of the original object. The Date class (§11.4) defines a `toJSON()` method that returns a serializable string representation of the date. We could do the same for our Point object like this:

```
let point = {
    x: 1,
    y: 2,
    toString: function() { return `(${this.x}, ${this.y})`; },
    toJSON: function() { return this.toString(); }
};
JSON.stringify([point])   // => '["(1, 2)"]'
```

6.10 Extended Object Literal Syntax

Recent versions of JavaScript have extended the syntax for object literals in a number of useful ways. The following subsections explain these extensions.

6.10.1 Shorthand Properties

Suppose you have values stored in variables `x` and `y` and want to create an object with properties named `x` and `y` that hold those values. With basic object literal syntax, you'd end up repeating each identifier twice:

```
let x = 1, y = 2;
let o = {
    x: x,
    y: y
};
```

In ES6 and later, you can drop the colon and one copy of the identifier and end up with much simpler code:

```
let x = 1, y = 2;
let o = { x, y };
o.x + o.y  // => 3
```

6.10.2 Computed Property Names

Sometimes you need to create an object with a specific property, but the name of that property is not a compile-time constant that you can type literally in your source code. Instead, the property name you need is stored in a variable or is the return value of a function that you invoke. You can't use a basic object literal for this kind of property. Instead, you have to create an object and then add the desired properties as an extra step:

```
const PROPERTY_NAME = "p1";
function computePropertyName() { return "p" + 2; }

let o = {};
o[PROPERTY_NAME] = 1;
o[computePropertyName()] = 2;
```

It is much simpler to set up an object like this with an ES6 feature known as *computed properties* that lets you take the square brackets from the preceding code and move them directly into the object literal:

```
const PROPERTY_NAME = "p1";
function computePropertyName() { return "p" + 2; }

let p = {
    [PROPERTY_NAME]: 1,
    [computePropertyName()]: 2
};

p.p1 + p.p2 // => 3
```

With this new syntax, the square brackets delimit an arbitrary JavaScript expression. That expression is evaluated, and the resulting value (converted to a string, if necessary) is used as the property name.

One situation where you might want to use computed properties is when you have a library of JavaScript code that expects to be passed objects with a particular set of properties, and the names of those properties are defined as constants in that library. If you are writing code to create the objects that will be passed to that library, you could hardcode the property names, but you'd risk bugs if you type the property name wrong anywhere, and you'd risk version mismatch issues if a new version of the library changes the required property names. Instead, you might find that it makes your code more robust to use computed property syntax with the property name constants defined by the library.

6.10.3 Symbols as Property Names

The computed property syntax enables one other very important object literal feature. In ES6 and later, property names can be strings or symbols. If you assign a symbol to a variable or constant, then you can use that symbol as a property name using the computed property syntax:

```
const extension = Symbol("my extension symbol");
let o = {
    [extension]: { /* extension data stored in this object */ }
};
o[extension].x = 0; // This won't conflict with other properties of o
```

As explained in §3.6, Symbols are opaque values. You can't do anything with them other than use them as property names. Every Symbol is different from every other Symbol, however, which means that Symbols are good for creating unique property names. Create a new Symbol by calling the `Symbol()` factory function. (Symbols are primitive values, not objects, so `Symbol()` is not a constructor function that you invoke with `new`.) The value returned by `Symbol()` is not equal to any other Symbol or other value. You can pass a string to `Symbol()`, and this string is used when your Symbol is converted to a string. But this is a debugging aid only: two Symbols created with the same string argument are still different from one another.

The point of Symbols is not security, but to define a safe extension mechanism for JavaScript objects. If you get an object from third-party code that you do not control and need to add some of your own properties to that object but want to be sure that your properties will not conflict with any properties that may already exist on the object, you can safely use Symbols as your property names. If you do this, you can also be confident that the third-party code will not accidentally alter your symbolically named properties. (That third-party code could, of course, use `Object.getOwnPropertySymbols()` to discover the Symbols you're using and could then alter or delete your properties. This is why Symbols are not a security mechanism.)

6.10.4 Spread Operator

In ES2018 and later, you can copy the properties of an existing object into a new object using the "spread operator" `...` inside an object literal:

```
let position = { x: 0, y: 0 };
let dimensions = { width: 100, height: 75 };
let rect = { ...position, ...dimensions };
rect.x + rect.y + rect.width + rect.height // => 175
```

In this code, the properties of the `position` and `dimensions` objects are "spread out" into the `rect` object literal as if they had been written literally inside those curly braces. Note that this `...` syntax is often called a spread operator but is not a true JavaScript operator in any sense. Instead, it is a special-case syntax available only within object literals. (Three dots are used for other purposes in other JavaScript contexts, but object literals are the only context where the three dots cause this kind of interpolation of one object into another one.)

If the object that is spread and the object it is being spread into both have a property with the same name, then the value of that property will be the one that comes last:

```
let o = { x: 1 };
let p = { x: 0, ...o };
p.x   // => 1: the value from object o overrides the initial value
let q = { ...o, x: 2 };
q.x   // => 2: the value 2 overrides the previous value from o.
```

Also note that the spread operator only spreads the own properties of an object, not any inherited ones:

```
let o = Object.create({x: 1}); // o inherits the property x
let p = { ...o };
p.x                            // => undefined
```

Finally, it is worth noting that, although the spread operator is just three little dots in your code, it can represent a substantial amount of work to the JavaScript interpreter. If an object has n properties, the process of spreading those properties into another object is likely to be an $O(n)$ operation. This means that if you find yourself using `...` within a loop or recursive function as a way to accumulate data into one large object, you may be writing an inefficient $O(n^2)$ algorithm that will not scale well as n gets larger.

6.10.5 Shorthand Methods

When a function is defined as a property of an object, we call that function a *method* (we'll have a lot more to say about methods in Chapters 8 and 9). Prior to ES6, you would define a method in an object literal using a function definition expression just as you would define any other property of an object:

```
let square = {
    area: function() { return this.side * this.side; },
    side: 10
};
square.area() // => 100
```

In ES6, however, the object literal syntax (and also the class definition syntax we'll see in Chapter 9) has been extended to allow a shortcut where the `function` keyword and the colon are omitted, resulting in code like this:

```
let square = {
    area() { return this.side * this.side; },
    side: 10
};
square.area() // => 100
```

Both forms of the code are equivalent: both add a property named `area` to the object literal, and both set the value of that property to the specified function. The shorthand syntax makes it clearer that `area()` is a method and not a data property like `side`.

When you write a method using this shorthand syntax, the property name can take any of the forms that are legal in an object literal: in addition to a regular JavaScript identifier like the name `area` above, you can also use string literals and computed property names, which can include Symbol property names:

```
const METHOD_NAME = "m";
const symbol = Symbol();
let weirdMethods = {
    "method With Spaces"(x) { return x + 1; },
    [METHOD_NAME](x) { return x + 2; },
    [symbol](x) { return x + 3; }
};
weirdMethods["method With Spaces"](1)  // => 2
```

```
weirdMethods[METHOD_NAME](1)            // => 3
weirdMethods[symbol](1)                 // => 4
```

Using a Symbol as a method name is not as strange as it seems. In order to make an object iterable (so it can be used with a `for/of` loop), you must define a method with the symbolic name `Symbol.iterator`, and there are examples of doing exactly that in Chapter 12.

6.10.6 Property Getters and Setters

All of the object properties we've discussed so far in this chapter have been *data properties* with a name and an ordinary value. JavaScript also supports *accessor properties*, which do not have a single value but instead have one or two accessor methods: a *getter* and/or a *setter*.

When a program queries the value of an accessor property, JavaScript invokes the getter method (passing no arguments). The return value of this method becomes the value of the property access expression. When a program sets the value of an accessor property, JavaScript invokes the setter method, passing the value of the righthand side of the assignment. This method is responsible for "setting," in some sense, the property value. The return value of the setter method is ignored.

If a property has both a getter and a setter method, it is a read/write property. If it has only a getter method, it is a read-only property. And if it has only a setter method, it is a write-only property (something that is not possible with data properties), and attempts to read it always evaluate to `undefined`.

Accessor properties can be defined with an extension to the object literal syntax (unlike the other ES6 extensions we've seen here, getters and setters were introduced in ES5):

```
let o = {
    // An ordinary data property
    dataProp: value,

    // An accessor property defined as a pair of functions.
    get accessorProp() { return this.dataProp; },
    set accessorProp(value) { this.dataProp = value; }
};
```

Accessor properties are defined as one or two methods whose name is the same as the property name. These look like ordinary methods defined using the ES6 shorthand except that getter and setter definitions are prefixed with `get` or `set`. (In ES6, you can also use computed property names when defining getters and setters. Simply replace the property name after `get` or `set` with an expression in square brackets.)

The accessor methods defined above simply get and set the value of a data property, and there is no reason to prefer the accessor property over the data property. But as a more interesting example, consider the following object that represents a 2D Cartesian point. It has ordinary data properties to represent the *x* and *y* coordinates of the point, and it has accessor properties that give the equivalent polar coordinates of the point:

```
let p = {
    // x and y are regular read-write data properties.
    x: 1.0,
    y: 1.0,

    // r is a read-write accessor property with getter and setter.
    // Don't forget to put a comma after accessor methods.
    get r() { return Math.hypot(this.x, this.y); },
    set r(newvalue) {
        let oldvalue = Math.hypot(this.x, this.y);
        let ratio = newvalue/oldvalue;
        this.x *= ratio;
        this.y *= ratio;
    },

    // theta is a read-only accessor property with getter only.
    get theta() { return Math.atan2(this.y, this.x); }
};
p.r     // => Math.SQRT2
p.theta // => Math.PI / 4
```

Note the use of the keyword `this` in the getters and setter in this example. JavaScript invokes these functions as methods of the object on which they are defined, which means that within the body of the function, `this` refers to the point object `p`. So the getter method for the `r` property can refer to the `x` and `y` properties as `this.x` and `this.y`. Methods and the `this` keyword are covered in more detail in §8.2.2.

Accessor properties are inherited, just as data properties are, so you can use the object `p` defined above as a prototype for other points. You can

give the new objects their own `x` and `y` properties, and they'll inherit the `r` and `theta` properties:

```
let q = Object.create(p); // A new object that inherits getters and setters
q.x = 3; q.y = 4;         // Create q's own data properties
q.r                       // => 5: the inherited accessor properties work
q.theta                   // => Math.atan2(4, 3)
```

The code above uses accessor properties to define an API that provides two representations (Cartesian coordinates and polar coordinates) of a single set of data. Other reasons to use accessor properties include sanity checking of property writes and returning different values on each property read:

```
// This object generates strictly increasing serial numbers
const serialnum = {
    // This data property holds the next serial number.
    // The _ in the property name hints that it is for internal use only.
    _n: 0,

    // Return the current value and increment it
    get next() { return this._n++; },

    // Set a new value of n, but only if it is larger than current
    set next(n) {
        if (n > this._n) this._n = n;
        else throw new Error("serial number can only be set to a larger valu
    }
};
serialnum.next = 10;    // Set the starting serial number
serialnum.next          // => 10
serialnum.next          // => 11: different value each time we get next
```

Finally, here is one more example that uses a getter method to implement a property with "magical" behavior:

```
// This object has accessor properties that return random numbers.
// The expression "random.octet", for example, yields a random number
// between 0 and 255 each time it is evaluated.
const random = {
    get octet() { return Math.floor(Math.random()*256); },
    get uint16() { return Math.floor(Math.random()*65536); },
    get int16() { return Math.floor(Math.random()*65536)-32768; }
};
```

6.11 Summary

This chapter has documented JavaScript objects in great detail, covering topics that include:

- Basic object terminology, including the meaning of terms like *enumerable* and *own property*.
- Object literal syntax, including the many new features in ES6 and later.
- How to read, write, delete, enumerate, and check for the presence of the properties of an object.
- How prototype-based inheritance works in JavaScript and how to create an object that inherits from another object with `Object.create()`.
- How to copy properties from one object into another with `Object.assign()`.

All JavaScript values that are not primitive values are objects. This includes both arrays and functions, which are the topics of the next two chapters.

1 Remember; almost all objects have a prototype but most do not have a property named `prototype`. JavaScript inheritance works even if you can't access the prototype object directly. But see §14.3 if you want to learn how to do that.