Chapter 6. Classes

JavaScript objects were covered in Chapter 3. That chapter treated each object as a unique set of properties, different from every other object. It is often useful, however, to define a *class* of objects that share certain properties. Members, or *instances*, of the class have their own properties to hold or define their state, but they also have methods that define their behavior. These methods are defined by the class and shared by all instances. Imagine a class named Complex that represents and performs arithmetic on complex numbers, for example. A Complex instance would have properties to hold the real and imaginary parts (the state) of the complex number. And the Complex class would define methods to perform addition and multiplication (the behavior) of those numbers.

In JavaScript, classes use prototype-based inheritance: if two objects inherit properties (generally function-valued properties, or methods) from the same prototype, then we say that those objects are instances of the same class. That, in a nutshell, is how JavaScript classes work. JavaScript prototypes and inheritance were covered in §3.2.3 and §3.3.2, and you will need to be familiar with the material in those sections to understand this chapter. This chapter covers prototypes in §6.1.

If two objects inherit from the same prototype, this typically (but not necessarily) means that they were created and initialized by the same constructor function or factory function.

JavaScript has always allowed the definition of classes. ECMAScript 6 introduced a brand-new syntax (including a `class` keyword) that makes it even easier to create classes. These new JavaScript classes work in the same way that old style classes do, and this chapter starts by explaining the old way of creating classes because that demonstrates more clearly what is going on behind the scenes to make classes work. Once we've explained those fundamentals, we'll shift and start using the new simplified class definition syntax.

If you're familiar with strongly-typed object-oriented programming languages like Java or C++, you'll notice that JavaScript classes are quite different from classes in those languages. There are some syntactic similarities, and you can emulate many features of "classical" classes in JavaScript, but it is best to understand up front that JavaScript's classes and prototype-based inheritance mechanism are substantially different from the classes and class-based inheritance mechanism of Java and similar languages.

6.1 Classes and Prototypes

In JavaScript, a class is a set of objects that inherit properties from the same prototype object. The prototype object, therefore, is the central feature of a class. Chapter 3 covered the `Object.create()` function that returns a newly created object that inherits from a specified prototype object. If we define a prototype object, and then use `Object.create()` to create objects that inherit from it, we have defined a JavaScript class. Usually, the instances of a class require further initialization, and it is common to define a function that creates and initializes the new object. Example 6-1 demonstrates this: it defines a prototype object for a class that represents a range of values and also defines a "factory" function that creates and initializes a new instance of the class.

Example 6-1. A simple JavaScript class

```
// This is a factory function that returns a new range object.
function range(from, to) {
    // Use Object.create() to create an object that inherits from the
    // prototype object defined below.  The prototype object is stored
    // a property of this function, and defines the shared methods (beh
    // for all range objects.
    let r = Object.create(range.methods);

    // Store the start and end points (state) of this new range object.
    // These are noninherited properties that are unique to this object
    r.from = from;
    r.to = to;

    // Finally return the new object
    return r;
}
```

```
// This prototype object defines methods inherited by all range objects
range.methods = {
    // Return true if x is in the range, false otherwise
    // This method works for textual and Date ranges as well as numeric
    includes(x) { return this.from <= x && x <= this.to; },

    // A generator function that makes instances of the class iterable.
    // Note that it only works for numeric ranges.
    *[Symbol.iterator]() {
        for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
    },

    // Return a string representation of the range
    toString() { return "(" + this.from + "..." + this.to + ")"; }
};

// Here are example uses of a range object.
let r = range(1,3);        // Create a range object
r.includes(2)              // => true: 2 is in the range
r.toString()               // => "(1...3)"
[...r]                     // => [1, 2, 3]; convert to an array via ite
```

There are a few things worth noting in the code of Example 6-1:

- This code defines a factory function `range()` for creating new range objects.
- It uses the `methods` property of this `range()` function as a convenient place to store the prototype object that defines the class. There is nothing special or idiomatic about putting the prototype object here.
- The `range()` function defines `from` and `to` properties on each range object. These are the unshared, noninherited properties that define the unique state of each individual range object.
- The `range.methods` object uses the ES6 shorthand syntax for defining methods, which is why you don't see the `function` keyword in the prototype. (See §3.10.5 to review object literal shorthand method syntax.)
- One of the methods in the prototype has the computed name (§3.10.2) `Symbol.iterator` which means that it is defining an iterator for range objects. The name of this method is prefixed with * which indicates that it is a generator function instead of a regular function. Iterators and generators are covered in detail in Chapter 7. For now, the upshot is that instances of this range class can be used with the `for/of` loop, and with the `...` spread operator.
- The shared, inherited methods defined in `range.methods` all use the `from` and `to` properties that were initialized in the `range()` factory function. In order to refer to them, they use the `this` keyword to refer to the object through which they were invoked. This use of `this` is a fundamental characteristic of the methods of any class.

6.2 Classes and Constructors

Example 6-1 demonstrates a simple way to define a JavaScript class. It is not the idiomatic way to do so, however, because it did not define a *constructor*. A constructor is a function designed for the initialization of newly created objects. Constructors are invoked using the `new` keyword as described in §5.2.3. Constructor invocations using `new` automatically create the new object, so the constructor itself only needs to initialize the state of that new object. The critical feature of constructor invocations is that the `prototype` property of the constructor is used as the prototype of the new object. This means that all objects created with the same constructor inherit from the same object and are therefore members of the same class. Example 6-2 shows how we could alter the range class of Example 6-1 to use a constructor function instead of a factory function. Example 6-2 demonstrates the idiomatic way to create a class in versions JavaScript that do not support the ES6 `class` keyword. Even though `class` is well-supported now, there is still lots of older JavaScript code around that defines classes like this, and you should be familiar with the idiom so that you can read old code and so that you understand what is going on "under the covers" when you use the `class` keyword.

Example 6-2. A Range class using a constructor

```
// This is a constructor function that initializes new Range objects.
// Note that it does not create or return the object. It just initializ
function Range(from, to) {
    // Store the start and end points (state) of this new range object.
    // These are noninherited properties that are unique to this object
    this.from = from;
    this.to = to;
}

// All Range objects inherit from this object.
// Note that the property name must be "prototype" for this to work.
Range.prototype = {
    // Return true if x is in the range, false otherwise
    // This method works for textual and Date ranges as well as numeric
    includes: function(x) { return this.from <= x && x <= this.to; },

    // A generator function that makes instances of the class iterable.
    // Note that it only works for numeric ranges.
    [Symbol.iterator]: function*() {
        for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
    },

    // Return a string representation of the range
    toString: function() { return "(" + this.from + "..." + this.to +
};
```

```
// Here are example uses of this new Range class
let r = new Range(1,3);   // Create a Range object
r.includes(2)             // => true: 2 is in the range
r.toString()              // => "(1...3)"
[...r]                    // => [1, 2, 3]; convert to an array via it
```

It is worth comparing Example 6-1 and Example 6-2 fairly carefully and noting the differences between these two techniques for defining classes. First, notice that we renamed the `range()` factory function to `Range()` when we converted it to a constructor. This is a very common coding convention: constructor functions define, in a sense, classes, and classes have names that begin with capital letters. Regular functions and methods have names that begin with lowercase letters.

Next, notice that the `Range()` constructor is invoked (at the end of the example) with the `new` keyword while the `range()` factory function was invoked without it. Example 6-1 uses regular function invocation (§5.2.1) to create the new object and Example 6-2 uses constructor invocation (§5.2.3). Because the `Range()` constructor is invoked with `new`, it does not have to call `Object.create()` or take any action to create a new object. The new object is automatically created before the constructor is called, and it is accessible as the `this` value. The `Range()` constructor merely has to initialize `this`. Constructors do not even have to return the newly created object. Constructor invocation automatically creates a new object, invokes the constructor as a method of that object, and returns the new object. The fact that constructor invocation is so different from regular function invocation is another reason that we give constructors names that start with capital letters. Constructors are written to be invoked as constructors, with the `new` keyword, and they usually won't work properly if they are invoked as regular functions. A naming convention that keeps constructor functions distinct from regular functions helps programmers to know when to use `new`.

Another critical difference between Example 6-1 and Example 6-2 is the way the prototype object is named. In the first example, the prototype was `range.methods`. This was a convenient and descriptive name, but arbitrary. In the second example, the prototype is `Range.prototype`, and this name is mandatory. An invocation of the `Range()` constructor automatically uses `Range.prototype` as the prototype of the new Range object.

Finally, also note the things that do not change between Example 6-1 and Example 6-2: the range methods are defined and invoked in the same way for both classes. Because Example 6-2 demonstrates the idiomatic way to create classes in versions of JavaScript before ES6, it does not use the ES6 shorthand method syntax in the prototype object and explicitly spells out the methods with the `function` keyword. But you can see that the implementation of the methods is the same in both examples.

Importantly, note that neither of the two range examples uses arrow functions when defining constructors or methods. Recall from §5.1.3 that functions defined in this way do not have a `prototype` property and so can not be used as constructors. Also, arrow function inherit the `this` keyword from the context in which they are defined rather than setting it based on the object which they are invoked through, and this makes them useless for methods, because the defining characteristic of methods is that they use `this` to refer to the instance on which they were invoked.

Fortunately, the new ES6 class syntax doesn't allow the option of defining methods with arrow functions, so this is not a mistake that you can accidentally make when using that syntax. We will cover the ES6 `class` keyword soon, but first there are more details to cover about constructors.

6.2.1 Constructors, Class Identity and `instanceof`

As we've seen, the prototype object is fundamental to the identity of a class: two objects are instances of the same class if and only if they inherit from the same prototype object. The constructor function that initializes the state of a new object is not fundamental: two constructor functions may have `prototype` properties that point to the same prototype object. Then both constructors can be used to create instances of the same class.

Even though constructors are not as fundamental as prototypes, the constructor serves as the public face of a class. Most obviously, the name of the constructor function is usually adopted as the name of the class. We say, for example, that the `Range()` constructor creates Range objects. More fundamentally, however, constructors are used with the `instanceof` operator when testing objects for membership in a class. If we have an object `r` and want to know if it is a Range object, we can write:

```
r instanceof Range   // => true: r inherits from Range.prototype
```

The `instanceof` operator was described earlier. The left-hand operand should be the object that is being tested, and the right-hand operand should be a constructor function that names a class. The expression `o instanceof c` evaluates to `true` if `o` inherits from `c.prototype`. The inheritance need not be direct. If `o` inherits from an object that inherits from an object that inherits from `c.prototype`, the expression will still evaluate to `true`.

In the code above, the `instanceof` operator is not actually checking whether `r` was initialized by the `Range` constructor. It is checking whether `r` inherits from

`Range.prototype`. Nevertheless, the `instanceof` syntax reinforces the use of constructors as the public identity of a class.

If you want to test the prototype chain of an object for a specific prototype and do not want to use the constructor function as an intermediary, you can use the `isPrototypeOf()` method. In Example 6-1, for example, we defined a class without a constructor function, so there is no way to use `instanceof` with that class. Instead, however, we could test whether an object `r` was a member of that constructor-less class with this code:

```
range.methods.isPrototypeOf(r);  // range.methods is the prototype obj
```

6.2.2 The constructor Property

In Example 6-2 we set `Range.prototype` to a new object that contained the methods for our class. Although it was convenient to express those methods as properties of a single object literal, it was not actually necessary to create a new object. Any regular JavaScript function (excluding arrow functions, generator functions and async functions) can be used as a constructor, and constructor invocations need a `prototype` property. Therefore, every regular JavaScript function[1] automatically has a `prototype` property. The value of this property is an object that has a single nonenumerable `constructor` property. The value of the `constructor` property is the function object:

```
let F = function() {}; // This is a function object.
let p = F.prototype;   // This is the prototype object associated with
let c = p.constructor; // This is the function associated with the pro
c === F                // => true: F.prototype.constructor === F for
```

The existence of this predefined prototype object with its `constructor` property means that objects typically inherit a `constructor` property that refers to their constructor. Since constructors serve as the public identity of a class, this constructor property gives the class of an object:

```
let o = new F();       // Create an object o of class F
o.constructor === F    // => true: the constructor property specifies th
```

Figure 6-1 illustrates this relationship between the constructor function, its prototype object, the back reference from the prototype to the constructor, and the instances created with the constructor.

[TODO: the line "foreach…" in the middle block of the figure should be removed, because I've removed it from the code]

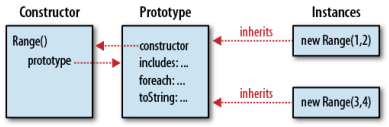

Figure 6-1. A constructor function, its prototype, and instances

Notice that Figure 6-1 uses our `Range()` constructor as an example. In fact, however, the Range class defined in Example 6-2 overwrites the predefined `Range.prototype` object with an object of its own. And the new prototype object it defines does not have a `constructor` property. So instances of the Range class, as defined, do not have a `constructor` property. We can remedy this problem by explicitly adding a constructor to the prototype:

```
Range.prototype = {
    constructor: Range,  // Explicitly set the constructor back-refere

    /* method definitions go here */
};
```

Another common technique that you are likely to see older JavaScript code is to use the predefined prototype object with its `constructor` property, and add methods to it one at a time with code like this:

```
// Extend the predefined Range.prototype object so we don't overwrite
// the automatically created Range.prototype.constructor property.
Range.prototype.includes = function(x) {
    return this.from<=x && x<=this.to;
};
Range.prototype.toString = function() {
    return "(" + this.from + "..." + this.to + ")";
};
```

6.3 Classes with the `class` Keyword

Classes have been part of JavaScript since the very first version of the language, but in ECMAScript 6 they finally got their own syntax, with the introduction of the `class` keyword. Example 6-3 shows what our Range class looks like when written with this new syntax.

Example 6-3. The Range class rewritten using `class`

```
class Range {
    constructor(from, to) {
        // Store the start and end points (state) of this new range ob
        // These are noninherited properties that are unique to this o
        this.from = from;
        this.to = to;
    }

    // Return true if x is in the range, false otherwise
    // This method works for textual and Date ranges as well as numeric
    includes(x) { return this.from <= x && x <= this.to; }

    // A generator function that makes instances of the class iterable.
    // Note that it only works for numeric ranges.
    *[Symbol.iterator]() {
        for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
    }

    // Return a string representation of the range
    toString() { return "(" + this.from + "..." + this.to + ")"; }
}

// Here are example uses of this new Range class
let r = new Range(1,3);   // Create a Range object
r.includes(2)             // => true: 2 is in the range
r.toString()              // => "(1...3)"
[...r]                    // => [1, 2, 3]; convert to an array via it
```

It is important to understand that the classes defined in Example 6-2 and Example 6-3 work in exactly the same way. The introduction of the `class` keyword to the language does not alter the fundamental nature of JavaScript's prototype-based classes. And although Example 6-3 uses the `class` keyword, the resulting `Range` object is a constructor function, just like the version defined in Example 6-2. The new `class` syntax is clean and convenient but is best thought of as "syntactic sugar" for the more fundamental class definition mechanism shown in Example 6-2.

Note the following things about the class syntax in Example 6-3:

- The class is declared with the `class` keyword which is followed by the name of class and a class body in curly braces.
- The class body includes method definitions that use object literal method shorthand (which we also used in Example 6-1) where the `function` keyword is omitted. In addition, however, no commas are used to separate the methods from each other.
- The keyword `constructor` is used to define the constructor function for the class. The function defined is not actually named "constructor", however. The `class` declaration statement defines a new variable `Range` and assigns the value of this special `constructor` function to that variable.
- If your class does not need to do any initialization, you can omit the `constructor` keyword and its body, and an empty constructor function will be implicitly created for you.

If you want to define a class that subclasses—or *inherits from*-another class, you can use the `extends` keyword with the `class` keyword:

```
// A Span is like a Range, but instead of initializing it with
// a start and an end, we initialize it with a start and a length
class Span extends Range {
    constructor(start, length) {
        if (length >= 0) {
            super(start, start + length);
        } else {
            super(start + length, start);
        }
    }
}
```

Creating subclasses is a whole topic of its own. We'll return to it, and explain the `extends` and `super` keywords shown here, in §6.5.

Like function declarations, class declarations have both statement and expression forms. Just as we can write:

```
let square = function(x) { return x * x; }
square(3)  // => 9
```

we can also write:

```
let Square = class { constructor(x) { this.area = x * x; } }
new Square(3).area  // => 9
```

As with function definition expressions, class definition expressions can include an optional class name. If you provide such a name, that name is only defined within the class body itself.

Although function expressions are quite common (particularly with the arrow function shorthand) in JavaScript programming, class definition expressions are not something that you are likely to use much unless you find yourself writing a function that takes a class as its argument and returns a subclass.

We'll conclude this introduction to the `class` keyword by mentioning a couple of important things you should know that are not apparent from `class` syntax:

- All code within the body of a `class` declaration is implicitly in strict mode even if no `"use strict"` directive appears. This means, for example, that you can't use octal integer literals or the `with` statement within class bodies, and that you are more likely to get syntax errors if you forget to declare a variable before using it.
- Unlike function declarations, class declarations are not "hoisted". Recall from earlier that function definitions behave as if they had been moved to the top of the enclosing file or enclosing function, meaning that you can invoke a function in code that comes before the actual definition of the function. Although class declarations are like function declarations in some ways, they do not share this hoisting behavior: you can *not* instantiate a class before you declare it.

6.3.1 Static Methods

You can define a static method within a `class` body by prefixing the method declaration with the `static` keyword. Static methods are defined as properties of the constructor function rather than properties of the prototype object. For example, suppose we added the following code to Example 6-3 above:

```
static parse(s) {
    let matches = s.match(/^\((\d+)\.\.\.(\d+)\)$/);
    if (!matches) {
        throw new TypeError(`Cannot parse Range from "${s}".`)
    }
    return new Range(parseInt(matches[1]), parseInt(matches[2]));
}
```

The method defined by this code is `Range.parse()`, not `Range.prototype.parse()`, and you must invoke it through the constructor, not through an instance:

```
let r = Range.parse('(1...10)'); // Returns a new Range object
r.parse('(1...10)');             // TypeError: r.parse is not a funct
```

You'll sometimes see static methods called *class methods* because they are invoked using the name of the class/constructor. When this term is used, it is to contrast class methods with the regular *instance methods* that are invoked on instances of the class. Because static methods are invoked on the constructor rather than on any particular instance, it almost never makes sense to use the `this` keyword in a static method.

We'll see examples of static methods in Example 6-4 below.

6.3.2 Getters, Setters and other Method Forms

Within a `class` body, you can define getter and setter methods (§3.10.6) just as you can in object literals. The only difference is that in class bodies, you don't put a comma after the getter or setter. Example 6-4 includes a practical example of a getter method in a class.

In general, the all of the shorthand method definition syntaxes allowed in object literals are also allowed in class bodies. This includes generator methods (marked with *), and methods whose names are the value of an expression in square brackets. In fact, you've already seen (in Example 6-3) a generator method with a computed name that makes the Range class iterable:

```
*[Symbol.iterator]() {
    for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
}
```

6.3.3 Public, Private and Static Fields

In the discussion above of classes defined with the `class` keyword, we have only described the definition of methods within the class body. The ES6 standard only allows the creation of methods (including getters, setters and generators) and static methods; it does not include syntax for defining fields. If you want to define a field (which is just another name for a property) on a class instance, you must do that in the constructor function or in one of the methods. And if you want to define a static field for a class, you must do that outside the class body, after the class has been defined. Example 6-4 includes examples of both kinds of fields.

Standardization is underway, however, for extended class syntax that allows the definition of instance and static fields, in both public and private forms. The code shown in the rest of this section is not legal JavaScript in early 2019 as I write this, but it is already supported in the Chrome web browser, is in fairly common use with transpilers like Babel, and may well have been standardized by the time you read this.

Suppose you're writing a class like this one, with a constructor that initializes three fields:

```
class Buffer {
    constructor() {
        this.size = 0;
        this.capacity = 4096;
        this.buffer = new Uint8Array(this.capacity);
    }
}
```

With the new instance field syntax that is likely to be standardized, you could instead write:

```
class Buffer {
    size = 0;
    capacity = 4096;
    buffer = new Uint8Array(this.capacity);
}
```

The field initialization code has moved out of the constructor and now appears directly in the class body. (That code is still run as part of the constructor, of course. If you do not define a constructor, the fields are initialized as part of the implicitly created constructor.) The `this.` prefixes that appeared on the left-hand side of the assignments are gone, but note, that you still must use `this.` to refer to these fields, even on the right-hand side of the initializer assignments. The advantage of initializing your instance fields in this way is that this syntax allows (but does not require) you to put the initializers up at the top of the class definition, making it clear to readers exactly what fields will hold the state of each instance. You can declare fields without an initializer by just writing the name of the field followed by a semicolon. If you do that the initial value of the field will be `undefined`. It is better style to always make the initial value explicit for all of your class fields.

Before the addition of this field syntax, class bodies looked a lot like object literals using shortcut method syntax, except that the commas had been removed. This field syntax, with equals signs and semicolons instead of colons and commas makes it clear, however that class bodies are not at all the same as object literals.

The same proposal that seeks to standardize these instance fields also defines private instance fields. If you use the instance field initialization syntax shown above to define a field whose name begins with `#` (which is not normally a legal character in JavaScript identifiers), that field will be usable (with the `#` prefix) within the class body, but will be invisible and inaccessible (and therefore immutable) to any code outside of the class body. If, for the hypothetical Buffer class above, you wanted to ensure that users of the class could not inadvertently modify the `size` field of an instance, you could use a private `#size` field instead, and then define a getter function to provide read-only access to the value:

```
class Buffer {
    #size = 0;
    get size() { return this.#size; }
}
```

Note that private fields must be declared using this new field syntax before they can be used. You can't just write `this.#size = 0;` in the constructor of a class unless you include a "declaration" of the field directly in the class body.

Finally, a related proposal seeks to standardize the use of the `static` keyword for fields. If you add `static` before a public or private field declaration, those fields will be created as properties of the constructor function instead of properties of instances. Consider the static `Range.parse()` method we defined above. It included a fairly complex regular expression that might be good to factor out into its own static field. With the proposed new static field syntax, we could do that like this:

```
static integerRangePattern = /^\((\d+)\.\.\.(\d+)\)$/;
static parse(s) {
    let matches = s.match(Range.integerRangePattern);
    if (!matches) {
        throw new TypeError(`Cannot parse Range from "${s}".`)
    }
    return new Range(parseInt(matches[1]), matches[2]);
}
```

If we wanted this static field to be accessible only within the class, we could make it private using a name like `#pattern`.

6.3.4 Example: a Complex Number Class

The example below defines a class to represent complex numbers. The class is a relatively simple one, but it includes instance methods (including getters), static methods, instance fields, and static fields. It includes some commented-out code demonstrating how we might use the not-yet-standard syntax for defining instance fields and static fields within the class body.

Example 6-4. Complex.js: A complex number class

```
/**
 * Instances of this Complex class represent complex numbers.
 * Recall that a complex number is the sum of a real number and an
 * imaginary number and that the imaginary number i is the square root
 */
class Complex {
    // Once class field declarations are standardized, we could declare
```

```
    // private fields to hold the real and imaginary parts of a comple
    // here, with code like this:
    //
    // #r = 0;
    // #i = 0;

    // This constructor function defines the instance fields r and i or
    // instance it creates. These fields hold the real and imaginary pa
    // the complex number: they are the state of the object.
    constructor(real, imaginary) {
        this.r = real;       // This field holds the real part of the
        this.i = imaginary;  // This field holds the imaginary part.
    }

    // Here are two instance methods for addition and multiplication
    // of complex numbers. If c and d are instances of this class, we
    // might write c.plus(d) or d.times(c)
    plus(that) {
        return new Complex(this.r + that.r, this.i + that.i);
    };
    times(that) {
        return new Complex(this.r * that.r - this.i * that.i,
                           this.r * that.i + this.i * that.r);
    };

    // And here are static variants of the complex arithmetic methods.
    // We could write Complex.sum(c,d) and Complex.product(c,d)
    static sum(c, d) { return c.plus(d); }
    static product(c, d) { return c.times(d); }

    // These are some instance methods that are defined as getters
    // so they're used like fields. The real and imaginary getters woul
    // be useful if we were using private fields this.#r and this.#i
    get real() { return this.r; }
    get imaginary() { return this.i; }
    get magnitude() { return Math.sqrt(this.r*this.r + this.i*this.i)

    // Classes should almost always have a toString() method
    toString() { return "{" + this.r + "," + this.i + "}"; }

    // It is often useful to define a method for testing whether
    // two instance of your class represent the same value
    equals(that) {
        return that instanceof Complex &&
            this.r === that.r &&
            this.i === that.i;
    }

    // Once static fields are supported inside class bodies, we could
    // define a useful Complex.ZERO constants like this:
    // static ZERO = new Complex(0,0);
}

// Here are some class fields that hold useful predefined complex numbe
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);
```

With the Complex class of Example 6-4 defined, we can use the constructor, instance fields, instance methods, class fields, and class methods with code like this:

```
let c = new Complex(2, 3);      // Create a new object with the constru
let d = new Complex(c.i, c.r); // Use instance fields of c
c.plus(d).toString()            // => "{5,5}"; use instance methods
c.magnitude                     // => Math.sqrt(13); use a getter func
Complex.product(c, d)           // => new Complex(0, 13); a static meth
Complex.ZERO.toString()         // => "{0,0}"; a static property
```

6.4 Adding Methods to Existing Classes

JavaScript's prototype-based inheritance mechanism is dynamic: an object inherits properties from its prototype, even if the properties of the prototype change after the object is created. This means that we can augment JavaScript classes simply by adding new methods to their prototype objects. Here, for example, is code that adds a method for computing the complex conjugate to the Complex class of Example 6-5:

```
// Return a complex number that is the complex conjugate of this one.
Complex.prototype.conj = function() { return new Complex(this.r, -this
```

The prototype object of built-in JavaScript classes is also open like this, which means that we can add methods to numbers, strings, arrays, functions, and so on. This is useful for implementing new language features in older versions of the language:

```
// If the new String method startsWith() is not already defined...
if (!String.prototype.startsWith) {
    // ...then define it like this using the older indexOf() method.
    String.prototype.startsWith = function(s) {
        return this.indexOf(s) === 0;
    };
}
```

Here is another example:

```
// Invoke the function f this many times, passing the iteration number
// For example, to print "hello" 3 times:
//     let n = 3;
//     n.times((i) => { console.log("hello"); });
```

```
Number.prototype.times = function(f, context) {
    let n = this.valueOf();
    for(let i = 0; i < n; i++) f.call(context, i);
};
```

Adding methods to the prototypes of built-in types like this is generally considered to be a bad idea because it will cause confusion and compatibility problems in the future if a new version of JavaScript defines a method with the same name. It is even possible to add methods to `Object.prototype`, making them available for all objects. But this is never a good thing to do because properties added to `Object.prototype` are visible to `for/in` loops (though you can avoid this if you use `Object.defineProperty()` to make the new property non-enumerable.)

In web browsers newer than Internet Explorer, you can add methods to `Document.prototype`, `HTMLElement.prototype` and other client-side JavaScript classes.

6.5 Subclasses

In object-oriented programming, a class B can *extend* or *subclass* another class A. We say that A is the *superclass* and B is the *subclass*. Instances of B inherit the methods of A. The class B can define its own methods, some of which may *override* methods of the same name defined by class A. If a method of B overrides a method of A, the overriding method in B often needs to invoke the overridden method in A. Similarly, the subclass constructor `B()` must typically invoke the superclass constructor `A()` in order to ensure that instances are completely initialized.

This section starts by showing how to define subclasses the old, pre-ES6 way, and then quickly moves on to demonstrate subclassing using the `class` and `extends` keywords, superclass constructor method invocation with the `super` keyword. Next is a sub-section about avoiding subclasses and relying on object composition instead of inheritance. The section ends with an extended example that defines a hierarchy of Set classes and demonstrates how abstract classes can be used to separate interface from implementation.

6.5.1 Subclasses and Prototypes

Suppose we wanted to define a Span subclass of the Range class from Example 6-2. This subclass will work just like a Range, but instead of initializing it with a start and an end, we'll instead specify a start and a distance, or span. An instance of this Span class is also an instance of the Range superclass. A span instance inherits a customized `toString()` method from `Span.prototype`, but in order to be a subclass of Range, it must also inherit methods (such as `includes()`) from `Range.prototype`.

Example 6-5. Span.js: A simple subclass of Range

```
// This is the constructor function for our subclass
function Span(start, span) {
    if (span >= 0) {
        this.from = start;
        this.to = start + span;
    } else {
        this.to = start;
        this.from = start + span;
    }
}

// Ensure that the Span prototype inherits from the Range prototype
Span.prototype = Object.create(Range.prototype);

// We don't want to inherit Range.prototype.constructor, so we
// define our own constructor property.
Span.prototype.constructor = Span;

// By defining its own toString() method, Span overrides the
// toString() method that it would otherwise inherit from Range.
Span.prototype.toString = function() {
    return `(${this.from}... +${this.to - this.from})`;
};
```

In order to make Span a subclass of Range, we need to arrange for `Span.prototype` to inherit from `Range.prototype`. The key line of code above is this one, and if it makes sense to you, you understand how subclasses work in JavaScript:

```
Span.prototype = Object.create(Range.prototype);
```

Objects created with the `Span()` constructor will inherit from the `Span.prototype` object. But we created that object to inherit from `Range.prototype`, so Span objects will inherit from both `Span.prototype` and `Range.prototype`.

You may notice that our `Span()` constructor sets the same `from` and `to` properties that the `Range()` constructor does and so does not need to invoke the `Range()` constructor to initialize the new object. Similarly, Span's `toString()` method completely re-implements the string conversion without needing to call Range's version of `toString()`. This makes Span a special case, and we can only really get away with this kind of subclassing because we know the implementation details of the superclass. A robust subclassing mechanism needs to allow classes to invoke the methods and constructor of their superclass, but prior to ES6, JavaScript did not have a simple way to do these things.

Fortunately, ES6 solves these problems with the `super` keyword as part of the `class` syntax. The next section demonstrates how it works.

6.5.2 Subclasses with `extends` and `super`

In ES6 and later, you can create a superclass simply by adding an `extends` clause to a class declaration, and you can do this even for built-in classes:

```
// A trivial Array subclass that adds getters for the first and last el
class EZArray extends Array {
    get first() { return this[0]; }
    get last() { return this[this.length-1]; }
};

let a = new EZArray();
a instanceof EZArray  // => true: a is subclass instance
a instanceof Array    // => true: a is also a superclass instance.
a.push(1,2,3,4);      // a.length == 4; we can use inherited methods
a.pop()               // => 4: another inherited method
a.first               // => 1: first getter defined by subclass
a.last                // => 3: last getter defined by subclass
a[1]                  // => 2: regular array access syntax still work
Array.isArray(a)      // => true: subclass instance really is an array
EZArray.isArray(a)    // => true: subclass inherits static methods, to
```

This EZArray subclass defines two simple getter methods. Instances of EZArray behave like ordinary arrays, and we can use inherited methods and properties like `push()`, `pop()`, and `length`. But we can also use the `first` and `last` getters defined in the subclass. Not only are instance methods like `pop()` inherited, but static methods like `Array.isArray` are also inherited. This is a new feature enabled by ES6 class syntax: `EZArray()` is a function, but it inherits from `Array()`:

```
// EZArray inherits instance methods because EZArray.prototype
// inherits from Array.prototype
Array.prototype.isPrototypeOf(EZArray.prototype) // => true

// And EZArray inherits static methods and properties because
// EZArray inherits from Array. This is a special feature of the
// extends keyword and is not possible before ES6.
Array.isPrototypeOf(EZArray) // => true
```

Our EZArray subclass is too simple to be very instructive. Example 6-6 is a more fully fleshed-out example. It defines a TypedMap subclass of the built-in Map class that adds type checking to ensure that the keys and values of the map are of the specified types (according to `typeof`). Importantly, this example demonstrates the use of the `super` keyword to invoke the constructor and methods of the superclass.

Example 6-6. TypedMap.js: A subclass of Map that checks key and value types

```
class TypedMap extends Map {
    constructor(keyType, valueType, entries) {
        // If entries are specified, check their types
        if (entries) {
            for(let [k, v] of entries) {
                if (typeof k !== keyType || typeof v !== valueType)
                    throw new TypeError(`Wrong type for entry [${k},
                }
            }
        }

        // Initialize the superclass with the (type-checked) initial e
        super(entries);

        // And then initialize this subclass by storing the types
        this.keyType = keyType;
        this.valueType = valueType;
    }

    // Now redefine the set() method to add type checking for any
    // new entries added to the map.
    set(key, value) {
        // Throw an error if the key or value are of the wrong type
        if (this.keyType && typeof key !== this.keyType) {
            throw new TypeError(`${key} is not of type ${this.keyType]
        }
        if (this.valueType && typeof value !== this.valueType) {
            throw new TypeError(`${value} is not of type ${this.value
        }

        // If the types are correct, we invoke the superclass's versio
        // the set() method, to actually add the entry to the map. And
        // return whatever the superclass method returns.
        return super.set(key, value);
    }
}
```

The first two arguments to the `TypedMap()` constructor are the desired key and value types. These should be strings, such as *number* and *boolean* that the `typeof` operator returns. You can also specify a third argument: an array (or any iterable object) of `[key,value]` arrays that specify the initial entries in the map. If you specify any initial entries, then the first thing the constructor does is verify that their types are correct. Next, the constructor invokes the superclass constructor, using the `super` keyword as if it was a function name. The `Map()` constructor takes one optional argument: an iterable object of `[key,value]` arrays. So the optional third argument of the `TypedMap()` constructor is the optional first argument to the `Map()` constructor, and we pass it to that superclass constructor with `super(entries)`.

After invoking the superclass constructor to initialize superclass state, the `TypedMap()` constructor next initializes its own subclass state by setting `this.keyType` and `this.valueType` to the specified types. It needs to set these properties so that it can use them again in the `set()` method.

There are a few important rules you need to know about using `super()` in constructors:

- If you define a class with the `extends` keyword, then the constructor for your class must use `super()` to invoke the superclass constructor.
- If you don't define a constructor in your subclass, one will be defined automatically for you. This implicitly defined constructor simply takes whatever values are passed to it and passes those values to `super()`
- You may not use the `this` keyword in your constructor until after you have invoked the superclass constructor with `super()`. This enforces a rule that superclasses get to initialize themselves before subclasses do.

After the constructor, the next part of Example 6-6 is a method named `set()`. The Map superclass defines a method named `set()` to add a new entry to the map. We say that this `set()` method in TypedMap *overrides* the `set()` method of its superclass. This simple TypedMap subclass doesn't know anything about adding new entries to map, but it does know how to check types, so that is what it does first, verifying that the key and value to be added to the map have the correct types, and throwing an error if they do not. This `set()` method doesn't have any way to add the key and value to the map itself, but that is what the superclass `set()` method is for. So we use the `super` keyword again to invoke the superclass's version of the method. In this context, `super` works much like the `this` keyword does: it refers to the current object, but allows access to overridden methods defined in the superclass.

In constructors you are required to invoke the superclass constructor before you can access `this` and initialize the new object yourself. There are no such rules when you override a method. A method that overrides a superclass method is not required to invoke the superclass method. If it does use `super` to invoke a the overridden method (or any method) in the superclass, it can do that at the beginning or the middle or the end of the overridding method.

Finally, before we leave the TypedMap example behind, it is worth noting that this class is an ideal candidate for the use of private fields. As the class is written now, a user could change the `keyType` or `valueType` properties to subvert the type-checking. Once private fields are supported, we could change these properties to `#keyType` and `#valueType` so that they could not be altered from the outside.

6.5.3 Delegation Instead of Inheritance

The `extends` keyword makes it easy to create subclasses. But that does not mean that you *should* create lots of subclasses. If you want to write a class that shares the behavior of some other class you can try to inherit that behavior by creating a subclass. But often it is easier and more flexible to get that desired behavior into your class by having your class create an instance of the other class and simply delegating to that instance as needed. You create a new class not by subclassing, but instead by wrapping or "composing" other classes. This delegation approach is often called "composition", and it is an oft-quoted maxim of object-oriented programming that one should "favor composition over inheritance"[2] .

Suppose, for example, we wanted a Histogram class that behaves something like JavaScript's Set class, except that instead of just keeping track of whether a value has been added the set or not, it instead maintains a count of the number of times the value has been added. Because the API for this Histogram class is similar to Set, we might consider subclassing Set, and adding a `count()` method. On the other hand, once we start thinking about how we might implement this `count()` method, we might realize that the Histogram class is more like a Map than a Set, because it needs to maintain a mapping between values and the number of times they have been added. So instead of subclassing Set, we can instead create a class that defines a Set-like API, but implements those methods by delegating to an internal Map object. Example 6-7 shows how we could do this:

Example 6-7. Histogram.js: A Set-like class implemented with delegation instead of inheritance

```
/**
 * A Set-like class that keeps track of how many times a value has
 * been added. Call add() and remove() like you would for a Set, and
 * call count() to find out how many times a given value has been added
 * The default iterator yields the values that have been added at least
 * once. Use entries() if you want to iterate [value, count] pairs.
 */
class Histogram {
    // To initialize, we just create a Map object to delegate to
    constructor() { this.map = new Map(); }

    // For any given key, the count is the value in the Map, or zero
    // if the key does not appear in the Map.
    count(key) { return this.map.get(key) || 0; }

    // The Set-like method has() returns true if the count is non-zero
    has(key) { return this.count(key) > 0; }

    // The size of the histogram is just the number of entries in the M
    get size() { return this.map.size; }

    // To add a key, just increment its count in the Map.
    add(key) { this.map.set(key, this.count(key) + 1); }
```

```
    // Deleting a key is a little trickier because we have to delete
    // the key from the Map if the count goes back down to zero.
    delete(key) {
        let count = this.count(key);
        if (count === 1) {
            this.map.delete(key);
        } else if (count > 1) {
            this.map.set(key, count - 1);
        }
    }

    // Iterating a Histogram just returns the keys stored in it
    [Symbol.iterator]() { return this.map.keys(); }

    // For compatibility with Set, the keys() and values() methods
    // do exactly the same thing as the default iterator.
    values() { return this.map.keys(); }
    keys() { return this.map.keys(); }

    // This Map-like method allows iteration of [value, count] pairs.
    entries() { return this.map.entries(); }
}
```

All the `Histogram()` constructor does in Example 6-7 is create a Map object. And most of the methods are one-liners that just delegate to a method of the map, making the implementation quite simple. Because we used delegation rather than inheritance, a Histogram object is not an instance of Set or Map. But Histogram implements a number of commonly-used Set methods, and in an untyped language like JavaScript that is often good enough: a formal inheritance relationship is sometimes nice, but often optional.

6.5.4 Class Hierarchies and Abstract Classes

Example 6-6 demonstrated how we can subclass Map. Example 6-7 demonstrated how we can instead delegate to a Map object without actually subclassing anything. Using JavaScript classes to encapsulate data and modularize your code is often a great technique, and you may find yourself using the `class` keyword frequently. But you may find that you prefer composition to inheritance and that you rarely need to use `extends` (except when you're using a library or framework that requires you to extend its base classes).

There are some circumstances when multiple levels of subclassing is appropriate however, and we'll end this chapter with an extended example that demonstrates a hierarchy of classes representing different kinds of sets. (The set classes defined in Example 6-8 are similar to, but not completely compatible with JavaScript's built-in Set class.)

Example 6-8 defines lots of subclasses, but it also demonstrates how you can define *abstract classes* – classes that do not include a complete implementation – to serve as a common superclass for a group of related subclasses. An abstract superclass can define a partial implementation that subclasses all inherit and share. The subclasses, then, only need to define their own unique behavior by implementing the abstract methods defined, but not implemented, by the superclass.

Example 6-8 is well-commented and stands on its own. I encourage you to read it as a capstone example for this chapter on classes. The final class in Example 6-8 does a lot of bit manipulation with the `&`, `|` and `~` operators.

Example 6-8. sets.js: A hierarchy of abstract and concrete set classes

```
/**
 * The AbstractSet class defines a single abstract method, has().
 */
class AbstractSet {
    has(x) { throw new Error("Abstract method"); }
}

/**
 * NotSet is a concrete subclass of AbstractSet.
 * The members of this set are all values that are not members of some
 * other set. Because it is defined in terms of another set it is not
 * writable, and because it has infinite members, it is not enumerable.
 * All we can do with it is test for membership and convert it to a
 * string using mathematical notation.
 */
class NotSet extends AbstractSet {
    constructor(set) {
        super();
        this.set = set;
    }

    // Our implementation of the abstract method we inherited
    has(x) { return !this.set.has(x); }
    // And we also override this Object method
    toString() { return `{ x| x ∉ ${this.set.toString()} }`; }
}

/**
 * Range set is a concrete subclass of AbstractSet. Its members are
 * all values that are between the from and to bounds, inclusive.
 * Since its members can be floating point numbers, it is not
 * enumerable and does not have a meaningful size.
 */
class RangeSet extends AbstractSet {
    constructor(from, to) {
        super();
        this.from = from;
        this.to = to;
    }

    has(x) { return x >= this.from && x <= this.to; }
    toString() { return `{ x| ${this.from} ≤ x ≤ ${this.to} }`; }
}
```

```
/*
 * AbstractEnumerableSet is an abstract subclass of AbstractSet.  It de
 * an abstract getter that returns the size of the set and also defines
 * abstract iterator. And it then implements concrete isEmpty(), toStri
 * and equals() methods on top of those. Subclasses that implement the
 * iterator, the size getter, and the has() method get these concrete
 * methods for free.
 */
class AbstractEnumerableSet extends AbstractSet {
    get size() { throw new Error("Abstract method"); }
    [Symbol.iterator]() { throw new Error("Abstract method"); }

    isEmpty() { return this.size === 0; }
    toString() { return `{${Array.from(this).join(', ')}}`; }
    equals(set) {
        // If the other set is not also Enumerable, it isn't equal to
        if (!(set instanceof AbstractEnumerableSet)) return false;

        // If they don't have the same size, they're not equal
        if (this.size != set.size) return false;

        // Loop through the elements of this set
        for(let element of this) {
            // If an element isn't in the other set, they aren't equal
            if (!set.has(element)) return false;
        }

        // The elements matched, so the sets are equal
        return true;
    }
}

/*
 * SingletonSet is a concrete subclass of AbstractEnumerableSet.
 * A singleton set is a read-only set with a single member.
 */
class SingletonSet extends AbstractEnumerableSet {
    constructor(member) {
        super();
        this.member = member;
    }

    // We implement these three methods, and inherit isEmpty, equals()
    // and toString() implementations based on these method.
    has(x) { return x === this.member; }
    get size() { return 1; }
    *[Symbol.iterator]() { yield this.member; }
}

/*
 * AbstractWritableSet is an abstract subclass of AbstractEnumerableSet
 * It defines the abstract methods insert() and remove() that insert an
 * remove individual elements from the set, and then implements concret
 * add(), subtract(), and intersect() methods on top of those. Note tha
 * our API diverges here from the standard JavaScript Set class.
 */
class AbstractWritableSet extends  AbstractEnumerableSet {
    insert(x) { throw new Error("Abstract method"); }
    remove(x) { throw new Error("Abstract method"); }

    add(set) {
        for(let element of set) {
            this.insert(element);
        }
    }

    subtract(set) {
        for(let element of set) {
            this.remove(element);
        }
    }

    intersect(set) {
        for(let element of this) {
            if (!set.has(element)) {
                this.remove(element);
            }
        }
    }
}

/**
 * A BitSet is a concrete subclass of AbstractWritableSet with a
 * very efficient fixed-size set implementation for sets whose
 * elements are non-negative integers less than some maximum size.
 */
class BitSet extends AbstractWritableSet {
    constructor(max) {
        super();
        this.max = max;  // The maximum integer we can store.
        this.n = 0;      // How many integers are in the set
        this.numBytes = Math.floor(max / 8) + 1;   // How many bytes
        this.data = new Uint8Array(this.numBytes); // The bytes
    }

    // Internal method to check if a value is a legal member of this se
    _valid(x) { return Number.isInteger(x) && x >= 0 && x <= this.max

    // Tests whether the specified bit of the specified byte of our
    // data array is set or not. Returns true or false.
    _has(byte, bit) { return (this.data[byte] & BitSet.bits[bit]) !==

    // Is the value x in this BitSet?
    has(x) {
        if (this._valid(x)) {
            let byte = Math.floor(x / 8);
            let bit = x % 8;
            return this._has(byte, bit);
        } else {
            return false;
        }
    }

    // Insert the value x into the BitSet
    insert(x) {
        if (this._valid(x)) {                // If the value is valid
            let byte = Math.floor(x / 8);    // convert to byte and bi
```

```
            let bit = x % 8;
            if (!this._has(byte, bit)) {     // If that bit is not set
                this.data[byte] |= BitSet.bits[bit]; // then set it
                this.n++;                            // and increme
            }
        } else {
            throw new TypeError("Invalid set element: " + x );
        }
    }

    remove(x) {
        if (this._valid(x)) {                // If the value is valid
            let byte = Math.floor(x / 8);  // compute the byte and bi
            let bit = x % 8;
            if (this._has(byte, bit)) {    // If that bit is already
                this.data[byte] &= BitSet.masks[bit];  // then unset
                this.n--;                              // and decre
            }
        } else {
            throw new TypeError("Invalid set element: " + x );
        }
    }

    // A getter to return the size of the set
    get size() { return this.n; }

    // Iterate the set by just checking each bit in turn.
    // (We could be a lot more clever and optimize this substantially)
    *[Symbol.iterator]() {
        for(let i = 0; i <= this.max; i++) {
            if (this.has(i)) {
                yield i;
            }
        }
    }
}

// Some pre-computed values used by the has(), insert() and remove() me
BitSet.bits = new Uint8Array([1, 2, 4, 8, 16, 32, 64, 128]);
BitSet.masks = new Uint8Array([~1, ~2, ~4, ~8, ~16, ~32, ~64, ~128]);
```

6.6 Summary

This chapter has explained the key features of JavaScript classes:

- Objects that are members of the same class inherit properties from the same prototype object. The prototype object is the key feature of JavaScript classes, and it is possible to define classes with nothing more than the `Object.create()` method.
- Prior to ES6, classes were more typically defined by first defining a constructor function. Functions created with the `function` keyword have a `prototype` property, and the value of this property is an object that is used as the prototype of all objects created when the function is invoked with `new` as a constructor. By initializing this prototype object you can define the shared methods of your class. Although the prototype object is still the key feature of the class, the constructor function is is the public identity of the class.
- ES6 introduces a `class` keyword that makes it easier to define classes, but under the hood, constructor and prototype mechanism remains the same.
- Subclasses are defined using the `extends` keyword in a class declaration.
- Subclasses can invoke the constructor of their superclass or overridden methods of their superclass with the `super` keyword.

---

1 Except functions returned by the ECMAScript 5 `Function.bind()` method. Bound functions have no prototype property of their own, but use the prototype of the underlying function if they are invoked as constructors.

2 See *Design Patterns* by Erich Gamma et al. or *Effective Java* by Joshua Bloch, for example.