Unit IV

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. In Example 6-1 we defined an inherit() function that returns a newly created object that inherits from a specified prototype object. If we define a prototype object, and then use inherit() 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 9-1 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 9-1. A simple JavaScript class

```
// range.js: A class representing a range of values.
// This is a factory function that returns a new range object.
function range(from, to) {
 // Use the inherit() function to create an object that inherits from the
 // prototype object defined below. The prototype object is stored as
 // a property of this function, and defines the shared methods (behavior)
 // for all range objects.
 var r = inherit(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: function(x) { return this.from <= x && x <= this.to; },
 // Invoke f once for each integer in the range.
 // This method works only for numeric ranges.
 foreach: function(f) {
 for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
 },
 // Return a string representation of the range
 toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};
// Here are example uses of a range object.
var r = range(1,3); // Create a range object
r.includes(2); // => true: 2 is in the range
r.foreach(console.log); // Prints 1 2 3
console.log(r); // Prints (1...3)
```

There are a few things worth noting in the code of Example 9-1. This code defines a factory function range() for creating new range objects. Notice that we use a property of this range() function range.methods 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. Second, notice that 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. Finally, notice that the shared, inherited methods defined in range.methods all use these from and to properties, and 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.

9.2 Classes and Constructors

Example 9-1 demonstrates one 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 §8.2.3. 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 9-2 shows how we could alter the range class of Example 9-1 to use a constructor function instead of a factory function:

Example 9-2. A Range class using a constructor

```
// range2.js: Another class representing a range of values.
// This is a constructor function that initializes new Range objects.
// Note that it does not create or return the object. It just initializes this.
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; },
 // Invoke f once for each integer in the range.
 // This method works only for numeric ranges.
 foreach: function(f) {
 for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
 },
 // Return a string representation of the range
 toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};
// Here are example uses of a range object
var r = new Range(1,3); // Create a range object
r.includes(2); // => true: 2 is in the range
r.foreach(console.log); // Prints 1 2 3
console.log(r); // Prints (1...3)
```

It is worth comparing Example 9-1 and Example 9-2 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 9-1 uses regular function invocation (§8.2.1) to create the new object and Example 9-2 uses constructor invocation (§8.2.3). Because the Range() constructor is invoked with new, it does not have to call inherit() 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.

Object-Oriented Techniques in JavaScript

So far in this chapter we've covered the architectural fundamentals of classes in JavaScript: the importance of the prototype object, its connections to the constructor function, how the instanceof operator works, and so on. In this section we switch gears and demonstrate a number of practical (though not fundamental) techniques for programming with JavaScript classes. We begin with two nontrivial example classes that are interesting in their own right but also serve as starting points for the discussions that follow.

A Set Class

A set is a data structure that represents an unordered collection of values, with no duplicates. The fundamental operations on sets are adding values and testing whether a value is a member of the set, and sets are generally implemented so that these operations are fast. JavaScript's objects are basically sets of property names, with values associated with each name. It is trivial, therefore, to use an object as a set of strings. Example 9-6 implements a more general Set class in JavaScript. It works by mapping any JavaScript value to a unique string, and then using that string as a property name. Objects and functions do not have a concise and reliably unique string representation,

so the Set class must define an identifying property on any object or function stored in the set.

Example 9-6. Set.js: An arbitrary set of values

```
function Set() { // This is the constructor
 this.values = {}; // The properties of this object hold the set
 this.n = 0; // How many values are in the set
 this.add.apply(this, arguments); // All arguments are values to add
}
// Add each of the arguments to the set.
Set.prototype.add = function() {
for(var i = 0; i < arguments.length; i++) { // For each argument
 var val = arguments[i]; // The value to add to the set
 var str = Set._v2s(val); // Transform it to a string
 if (!this.values.hasOwnProperty(str)) { // If not already in the set
 this.values[str] = val; // Map string to value
 this.n++; // Increase set size
 }
 }
 return this; // Support chained method calls
};
// Remove each of the arguments from the set.
Set.prototype.remove = function() {
 for(var i = 0; i < arguments.length; i++) { // For each argument
 var str = Set._v2s(arguments[i]); // Map to a string
 if (this.values.hasOwnProperty(str)) { // If it is in the set
 delete this.values[str]; // Delete it
 this.n--; // Decrease set size
```

```
 }
 }
 return this; // For method chaining
};
// Return true if the set contains value; false otherwise.
Set.prototype.contains = function(value) {
 return this.values.hasOwnProperty(Set._v2s(value));
};
// Return the size of the set.
Set.prototype.size = function() { return this.n; };
// Call function f on the specified context for each element of the set.
Set.prototype.foreach = function(f, context) {
 for(var s in this.values) // For each string in the set
 if (this.values.hasOwnProperty(s)) // Ignore inherited properties
 f.call(context, this.values[s]); // Call f on the value
};
// This internal function maps any JavaScript value to a unique string.
Set._v2s = function(val) {
 switch(val) {
 case undefined: return 'u'; // Special primitive
 case null: return 'n'; // values get single-letter
 case true: return 't'; // codes.
 case false: return 'f';
 default: switch(typeof val) {
 case 'number': return '#' + val; // Numbers get # prefix.
 case 'string': return '"' + val; // Strings get " prefix.
 default: return '@' + objectId(val); // Objs and funcs get @
```

```
 }
 }
 // For any object, return a string. This function will return a different
 // string for different objects, and will always return the same string
 // if called multiple times for the same object. To do this it creates a
 // property on o. In ES5 the property would be nonenumerable and read-only.
function objectId(o) {
 var prop = "|**objectid**|"; // Private property name for storing ids
 if (!o.hasOwnProperty(prop)) // If the object has no id
 o[prop] = Set._v2s.next++; // Assign it the next available
 return o[prop]; // Return the id
 }
};
Set._v2s.next = 100; // Start assigning object ids at this value.
```

.2 Example: Enumerated Types

An enumerated type is a type with a finite set of values that are listed (or “enumerated”) when the type is defined. In C and languages derived from it, enumerated types are declared with the enum keyword. enum is a reserved (but unused) word in ECMAScript 5 which leaves open the possibility that JavaScript may someday have native enumerated types. Until then, Example 9-7 shows how you can define your own enumerated types in JavaScript. Note that it uses the inherit() function from Example 6-1.

Example 9-7 consists of a single function enumeration(). This is not a constructor function, however: it does not define a class named “enumeration”. Instead, this is a factory function: each invocation creates and returns a new class. Use it like this:

```
// Create a new Coin class with four values: Coin.Penny, Coin.Nickel, etc.
var Coin = enumeration({Penny: 1, Nickel:5, Dime:10, Quarter:25});
```

```
var c = Coin.Dime; // This is an instance of the new class
c instanceof Coin // => true: instanceof works
c.constructor == Coin // => true: constructor property works
Coin.Quarter + 3*Coin.Nickel // => 40: values convert to numbers
Coin.Dime == 10 // => true: more conversion to numbers
Coin.Dime > Coin.Nickel // => true: relational operators work
String(Coin.Dime) + ":" + Coin.Dime // => "Dime:10": coerce to string
```

The point of this example is to demonstrate that JavaScript classes are much more flexible and dynamic than the static classes of languages like C++ and Java.

Example 9-7. Enumerated types in JavaScript

```
// This function creates a new enumerated type. The argument object specifies
// the names and values of each instance of the class. The return value
// is a constructor function that identifies the new class. Note, however
// that the constructor throws an exception: you can't use it to create new
// instances of the type. The returned constructor has properties that
// map the name of a value to the value itself, and also a values array,
// a foreach() iterator function
function enumeration(namesToValues) {
 // This is the dummy constructor function that will be the return value.
 var enumeration = function() { throw "Can't Instantiate Enumerations"; };
 // Enumerated values inherit from this object.
 var proto = enumeration.prototype = {
 constructor: enumeration, // Identify type
 toString: function() { return this.name; }, // Return name
 valueOf: function() { return this.value; }, // Return value
toJSON: function() { return this.name; } // For serialization
 };
```

```
 enumeration.values = []; // An array of the enumerated value objects
 // Now create the instances of this new type.
 for(name in namesToValues) { // For each value
 var e = inherit(proto); // Create an object to represent it
 e.name = name; // Give it a name
 e.value = namesToValues[name]; // And a value
 enumeration[name] = e; // Make it a property of constructor
 enumeration.values.push(e); // And store in the values array
 }
 // A class method for iterating the instances of the class
 enumeration.foreach = function(f,c) {
 for(var i = 0; i < this.values.length; i++) f.call(c,this.values[i]);
 };
 // Return the constructor that identifies the new type
 return enumeration;
}
```

The “hello world” of enumerated types is to use an enumerated type to represent the suits in a deck of cards. Example 9-8 uses the enumeration() function in this way and also defines classes to represents cards and decks of cards.1

Example 9-8. Representing cards with enumerated types

```
// Define a class to represent a playing card
function Card(suit, rank) {
 this.suit = suit; // Each card has a suit
 this.rank = rank; // and a rank
}
// These enumerated types define the suit and rank values
Card.Suit = enumeration({Clubs: 1, Diamonds: 2, Hearts:3, Spades:4});
```

```
Card.Rank = enumeration({Two: 2, Three: 3, Four: 4, Five: 5, Six: 6,
 Seven: 7, Eight: 8, Nine: 9, Ten: 10,
 Jack: 11, Queen: 12, King: 13, Ace: 14});
// Define a textual representation for a card
Card.prototype.toString = function() {
 return this.rank.toString() + " of " + this.suit.toString();
};
// Compare the value of two cards as you would in poker
Card.prototype.compareTo = function(that) {
 if (this.rank < that.rank) return -1;
 if (this.rank > that.rank) return 1;
 return 0;
};
// A function for ordering cards as you would in poker
1. This example is based on a Java example by Joshua Bloch, available at http://jcp.org/aboutJava/
communityprocess/jsr/tiger/enum.html.
Card.orderByRank = function(a,b) { return a.compareTo(b); };
// A function for ordering cards as you would in bridge
Card.orderBySuit = function(a,b) {
 if (a.suit < b.suit) return -1;
 if (a.suit > b.suit) return 1;
 if (a.rank < b.rank) return -1;
 if (a.rank > b.rank) return 1;
 return 0;
};
// Define a class to represent a standard deck of cards
function Deck() {
```

```
 var cards = this.cards = []; // A deck is just an array of cards
 Card.Suit.foreach(function(s) { // Initialize the array
 Card.Rank.foreach(function(r) {
 cards.push(new Card(s,r));
 });
 });
}
// Shuffle method: shuffles cards in place and returns the deck
Deck.prototype.shuffle = function() {
 // For each element in the array, swap with a randomly chosen lower element
 var deck = this.cards, len = deck.length;
 for(var i = len-1; i > 0; i--) {
 var r = Math.floor(Math.random()*(i+1)), temp; // Random number
 temp = deck[i], deck[i] = deck[r], deck[r] = temp; // Swap
 }
 return this;
};
// Deal method: returns an array of cards
Deck.prototype.deal = function(n) {
 if (this.cards.length < n) throw "Out of cards";
 return this.cards.splice(this.cards.length-n, n);
};
// Create a new deck of cards, shuffle it, and deal a bridge hand
var deck = (new Deck()).shuffle();
var hand = deck.deal(13).sort(Card.orderBySuit);
```

9.6.3 Standard Conversion Methods

§3.8.3 and §6.10 described important methods used for type conversion of objects,

some of which are invoked automatically by the JavaScript interpreter when conversion is necessary. You do not need to implement these methods for every class you write, but they are important methods, and if you do not implement them for your classes, it should be a conscious choice not to implement them rather than mere oversight.

The first, and most important, method is toString(). The purpose of this method is to return a string representation of an object. JavaScript automatically invokes this method if you use an object where a string is expected—as a property name, for example, or with the + operator to perform string concatenation. If you don't implement this method, your class will inherit the default implementation from Object.prototype and will convert to the useless string "[object Object]". A toString() method might return a human-readable string suitable for display to end users of your program. Even if this is not necessary, however, it is often useful to define toString() for ease of debugging. The Range and Complex classes in Examples 9-2 and 9-3 have toString() methods, as do the enumerated types of Example 9-7. We'll define a toString() method for the Set class of Example 9-6 below.

The toLocaleString() is closely related to toString(): it should convert an object to a string in a locale-sensitive way. By default, objects inherit a toLocaleString() method that simply calls their toString() method. Some built-in types have useful toLocaleString() methods that actually return locale-dependent strings. If you find yourself writing a toString() method that converts other objects to strings, you should also define a toLocaleString() method that performs those conversions by invoking the toLocaleString() method on the objects. We'll do this for the Set class below.

The third method is valueOf(). Its job is to convert an object to a primitive value. The valueOf() method is invoked automatically when an object is used in a numeric context, with arithmetic operators (other than +) and with the relational operators, for example. Most objects do not have a reasonable primitive representation and do not define this method. The enumerated types in Example 9-7 demonstrate a case in which the

valueOf() method is important, however.

The fourth method is toJSON(), which is invoked automatically by JSON.stringify(). The JSON format is intended for serialization of data structures and can handle JavaScript primitive values, arrays, and plain objects. It does not know about classes, and when serializing an object, it ignores the object's prototype and constructor. If you call JSON.stringify() on a Range or Complex object, for example, it returns a string like {"from":1, "to":3} or {"r":1, "i":-1}. If you pass these strings to JSON.parse(), you'll obtain a plain object with properties appropriate for Range and Complex objects, but which do not inherit the Range and Complex methods.

This kind of serialization is appropriate for classes like Range and Complex, but for other classes you may want to write a toJSON() method to define some other serialization format. If an object has a toJSON() method, JSON.stringify() does not serialize the object but instead calls toJSON() and serializes the value (either primitive or object) that it returns. Date objects, for example, have a toJSON() method that returns a string representation of the date. The enumerated types of Example 9-7 do the same: their toJSON() method is the same as their toString() method. The closest JSON analog to a set is an array, so we'll define a toJSON() method below that converts a Set object to an array of values.

The Set class of Example 9-6 does not define any of these methods. A set has no primitive representation, so it doesn't make sense to define a valueOf() method, but the class should probably have toString(), toLocaleString(), and toJSON() methods. We can do that with code like the following. Note the use of the extend() function (Example 6-2) to add methods to Set.prototype:

```
// Add these methods to the Set prototype object.
extend(Set.prototype, {
 // Convert a set to a string
 toString: function() {
```

```
var s = "{", i = 0;
this.foreach(function(v) { s += ((i++ > 0)?", ":"") + v; });
return s + "}";
},
// Like toString, but call toLocaleString on all values
toLocaleString : function() {
var s = "{", i = 0;
this.foreach(function(v) {
if (i++ > 0) s += ", ";
if (v == null) s += v; // null & undefined
else s += v.toLocaleString(); // all others
});
return s + "}";
},
// Convert a set to an array of values
toArray: function() {
var a = [];
this.foreach(function(v) { a.push(v); });
return a;
}
});
// Treat sets like arrays for the purposes of JSON stringification.
Set.prototype.toJSON = Set.prototype.toArray;
```

9.6.4 Comparison Methods

JavaScript equality operators compare objects by reference, not by value. That is, given two object references, they look to see if both references are to the same object. They do not check to see if two different objects have the same property names and values.

It is often useful to be able to compare two distinct objects for equality or even for relative order (as the

< and

> operators do). If you define a class and want to be able to

compare instances of that class, you should define appropriate methods to perform those comparisons.

The Java programming language uses methods for object comparison, and adopting the Java conventions is a common and useful thing to do in JavaScript. To enable instances of your class to be tested for equality, define an instance method named equals(). It should take a single argument and return true if that argument is equal to the object it is invoked on. Of course it is up to you to decide what “equal” means in the context of your own class. For simple classes you can often simply compare the constructor properties to ensure that the two objects are of the same type and then compare the instance properties of the two objects to ensure that they have the same values. The Complex class in Example 9-3 has an equals() method of this sort, and we can easily write a similar one for the Range class:

```
// The Range class overwrote its constructor property. So add it now.
Range.prototype.constructor = Range;
// A Range is not equal to any nonrange.
// Two ranges are equal if and only if their endpoints are equal.
Range.prototype.equals = function(that) {
 if (that == null) return false; // Reject null and undefined
 if (that.constructor !== Range) return false; // Reject non-ranges
 // Now return true if and only if the two endpoints are equal.
 return this.from == that.from && this.to == that.to;
}
```

Defining an equals() method for our Set class is somewhat trickier. We can’t just com-

pare the values property of two sets but must perform a deeper comparison:

```
Set.prototype.equals = function(that) {
 // Shortcut for trivial case
 if (this === that) return true;
 // If the that object is not a set, it is not equal to this one.
 // We use instanceof to allow any subclass of Set.
 // We could relax this test if we wanted true duck-typing.
 // Or we could strengthen it to check this.constructor == that.constructor
 // Note that instanceof properly rejects null and undefined values
 if (!(that instanceof Set)) return false;
 // If two sets don't have the same size, they're not equal
 if (this.size() != that.size()) return false;
 // Now check whether every element in this is also in that.
 // Use an exception to break out of the foreach if the sets are not equal.
 try {
 this.foreach(function(v) { if (!that.contains(v)) throw false; });
 return true; // All elements matched: sets are equal.
 } catch (x) {
 if (x === false) return false; // An element in this is not in that.
 throw x; // Some other exception: rethrow it.
 }
};
```

It is sometimes useful to compare objects according to some ordering. That is, for some classes, it is possible to say that one instance is “less than” or “greater than” another instance. You might order Range object based on the value of their lower bound, for example. Enumerated types could be ordered alphabetically by name, or numerically by the associated value (assuming the associated value is a number). Set objects, on the

other hand, do not really have a natural ordering.

If you try to use objects with JavaScript's relation operators, such as < and <=, JavaScript first calls the valueOf() method of the objects and, if this method returns a primitive value, compares those values. The enumerated types returned by the enumeration() method of Example 9-7 have a valueOf() method and can be meaningfully compared using the relational operators. Most classes do not have a valueOf() method, however. To compare objects of these types according to an explicitly defined ordering of your own choosing, you can (again, following Java convention) define a method named compareTo().

The compareTo() method should accept a single argument and compare it to the object on which the method is invoked. If the this object is less than the argument, compareTo() should return a value less than zero. If the this object is greater than the argument object, the method should return a value greater than zero. And if the two objects are equal, the method should return zero. These conventions about the return value are important, and they allow you to substitute the following expressions for relational and equality operators:

Replace this With this

a < b a.compareTo(b) < 0

a <= b a.compareTo(b) <= 0

a > b a.compareTo(b) > 0

a >= b a.compareTo(b) >= 0

a == b a.compareTo(b) == 0

a != b a.compareTo(b) != 0

The Card class of Example 9-8 defines a compareTo() method of this kind, and we can write a similar method for the Range class to order ranges by their lower bound:

```
Range.prototype.compareTo = function(that) {
 return this.from - that.from;
```

};

Notice that the subtraction performed by this method correctly returns a value less than zero, equal to zero, or greater than zero, according to the relative order of the two Ranges. Because the Card.Rank enumeration in Example 9-8 has a valueOf() method, we could have used this same idiomatic trick in the compareTo() method of the Card class.

The equals() methods above perform type checking on their argument and return false to indicate inequality if the argument is of the wrong type. The compareTo() method does not have any return value that indicates "those two values are not comparable," so a compareTo() method that does type checking should typically throw an error when passed an argument of the wrong type.

Notice that the compareTo() method we defined for the Range class above returns 0 when two ranges have the same lower bound. This means that as far as compareTo() is concerned, any two ranges that start at the same spot are equal. This definition of equality is inconsistent with the definition used by the equals() method, which requires both endpoints to match. Inconsistent notions of equality can be a pernicious source of bugs, and it is best to make your equals() and compareTo() methods consistent. Here is a revised compareTo() method for the Range class. It is consistent with equals() and also throws an error if called with an incomparable value:

Object constructors and prototyping :

Constructor Overloading and Factory Methods

Sometimes we want to allow objects to be initialized in more than one way. We might want to create a Complex object initialized with a radius and an angle (polar coordinates) instead of real and imaginary components, for example, or we might want to create a Set whose members are the elements of an array rather than the arguments passed to the constructor.

One way to do this is to overload the constructor and have it perform different kinds

of initialization depending on the arguments it is passed. Here is an overloaded version of the Set constructor, for example:

```
function Set() {
 this.values = {}; // The properties of this object hold the set
 this.n = 0; // How many values are in the set
 // If passed a single array-like object, add its elements to the set
 // Otherwise, add all arguments to the set
 if (arguments.length == 1 && isArrayLike(arguments[0]))
 this.add.apply(this, arguments[0]);
 else if (arguments.length > 0)
 this.add.apply(this, arguments);
}
```

Defining the Set() constructor this way allows us to explicitly list set members in the constructor call or to pass an array of members to the constructor. The constructor has an unfortunate ambiguity, however: we cannot use it to create a set that has an array as its sole member. (To do that, we'd have to create an empty set and then call the add() method explicitly.)

In the case of complex numbers initialized to polar coordinates, constructor overloading really isn't viable. Both representations of complex numbers involve two floating-point numbers and, unless we add a third argument to the constructor, there is no way for the constructor to examine its arguments and determine which representation is desired. Instead, we can write a factory method—a class method that returns an instance of the class. Here is a factory method for returning a Complex object initialized using polar coordinates:

```
Complex.polar = function(r, theta) {
 return new Complex(r*Math.cos(theta), r*Math.sin(theta));
};
```

And here is a factory method for initializing a Set from an array:

```
Set.fromArray = function(a) {
 s = new Set(); // Create a new empty set
 s.add.apply(s, a); // Pass elements of array a to the add method
return s; // Return the new set
};
```

The appeal of factory methods here is that you can give them whatever name you want, and methods with different names can perform different kinds of initializations. Since constructors serve as the public identity of a class, however, there is usually only a single constructor per class. This is not a hard-and-fast rule, however. In JavaScript it is possible to define multiple constructor functions that share a single prototype object, and if you do this, objects created by any of the constructors will be of the same type. This technique is not recommended, but here is an auxiliary constructor of this type:

```
// An auxiliary constructor for the Set class.
function SetFromArray(a) {
 // Initialize new object by invoking Set() as a function,
 // passing the elements of a as individual arguments.
 Set.apply(this, a);
}
// Set the prototype so that SetFromArray creates instances of Set
SetFromArray.prototype = Set.prototype;
var s = new SetFromArray([1,2,3]);
s instanceof Set // => true
```

In ECMAScript 5, the bind() method of functions has special behavior that allows it to create this kind of auxiliary constructor. See §8.7.4.

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 all the instance methods of A. The class B can define its own instance 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 may sometimes want to invoke the overridden method in A: this is called method chaining. Similarly, the subclass constructor B() may sometimes need to invoke the superclass constructor A(). This is called constructor chaining. Subclasses can themselves have subclasses, and when working with hierarchies of classes, it can sometimes be useful to define abstract classes. An abstract class is one that defines one or more methods without an implementation. The implementation of these abstract methods is left to the concrete subclasses of the abstract class.

The key to creating subclasses in JavaScript is proper initialization of the prototype object. If class B extends A, then B.prototype must be an heir of A.prototype. Then instances of B will inherit from B.prototype which in turn inherits from A.prototype. This section demonstrates each of the subclass-related terms defined above, and also covers an alternative to subclassing known as composition.

Using the Set class of Example 9-6 as a starting point, this section will demonstrate how to define subclasses, how to chain to constructors and overridden methods, how to use composition instead of inheritance, and finally, how to separate interface from imple mentation with abstract classes. The section ends with an extended example that defines a hierarchy of Set classes. Note that the early examples in this section are intended to demonstrate basic subclassing techniques. Some of these examples have important flaws that will be addressed later in the section.

9.7.1 Defining a Subclass

JavaScript objects inherit properties (usually methods) from the prototype object of their class. If an object O is an instance of a class B and B is a subclass of A, then O must also inherit properties from A. We arrange this by ensuring that the prototype object of B inherits from the prototype object of A. Using our inherit() function

(Example 6-1), we write:

```
B.prototype = inherit(A.prototype); // Subclass inherits from superclass
B.prototype.constructor = B; // Override the inherited constructor prop.
```

These two lines of code are the key to creating subclasses in JavaScript. Without them, the prototype object will be an ordinary object—an object that inherits from Object.prototype—and this means that your class will be a subclass of Object like all classes are. If we add these two lines to the defineClass() function (from §9.3), we can transform it into the defineSubclass() function and the Function.proto type.extend() method shown in Example 9-11.

Example 9-11. Subclass definition utilities

```
// A simple function for creating simple subclasses
function defineSubclass(superclass, // Constructor of the superclass
 constructor, // The constructor for the new subclass
 methods, // Instance methods: copied to prototype
 statics) // Class properties: copied to constructor
{
 // Set up the prototype object of the subclass
 constructor.prototype = inherit(superclass.prototype);
 constructor.prototype.constructor = constructor;
 // Copy the methods and statics as we would for a regular class
 if (methods) extend(constructor.prototype, methods);
 if (statics) extend(constructor, statics);
 // Return the class
 return constructor;
}
// We can also do this as a method of the superclass constructor
Function.prototype.extend = function(constructor, methods, statics) {
```

return defineSubclass(this, constructor, methods, statics);

};

JSON

Since we are coming from JavaScript, it might be convenient to use a JavaScript friendly data format as our transport: enter JSON (JavaScript Object Notation), defined at www.json.org. JSON is a lightweight data-interchange format that is based on a subset of the JavaScript language. However, it is actually pretty much language independent and can easily be consumed by various server-side languages.

The values that are allowed in the JSON format are strings in double quotes like "Thomas"; numbers like 1, 345.7, or 1.07E4; the values true, false, and null; or an array or object. The syntax trees from JSON.org show the format clearly so we present those here with a brief discussion and set of examples.

JSON - Syntax

Let's have a quick look at the basic syntax of JSON. JSON syntax is basically considered as a subset of JavaScript syntax; it includes the following –

Data is represented in name/value pairs.

Curly braces hold objects and each name is followed by ':'(colon), the name/value pairs are separated by , (comma).

Square brackets hold arrays and values are separated by ,(comma).

Below is a simple example –

```
{
   "book": [
```

```
    {
      "id": "01",
      "language": "Java",
      "edition": "third",
      "author": "Herbert Schildt"
    },

    {
      "id": "07",
      "language": "C++",
      "edition": "second",
      "author": "E.Balagurusamy"
    }

  ]
}
```

JSON supports the following two data structures −

Collection of name/value pairs − This Data Structure is supported by different programming languages.

Ordered list of values − It includes array, list, vector or sequence etc.

J query introduction :

jQuery is a fast and concise JavaScript Library created by John Resig in 2006 with a nice motto: Write less, do more. jQuery simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is a JavaScript toolkit designed to simplify various tasks by writing less code. Here is the list of important core features supported by jQuery −

DOM manipulation − The jQuery made it easy to select DOM elements, negotiate them and modifying their content by using cross-browser open source selector engine called Sizzle.

Event handling − The jQuery offers an elegant way to capture a wide variety of events, such as a user clicking on a link, without the need to clutter the HTML code itself with event handlers.

AJAX Support − The jQuery helps you a lot to develop a responsive and featurerich site using AJAX technology.

Animations − The jQuery comes with plenty of built-in animation effects which you can use in your websites.

Lightweight − The jQuery is very lightweight library - about 19KB in size (Minified and gzipped).

Cross Browser Support − The jQuery has cross-browser support, and works well in IE 6.0+, FF 2.0+, Safari 3.0+, Chrome and Opera 9.0+

Latest Technology − The jQuery supports CSS3 selectors and basic XPath syntax.

Ajax introduction :

AJAX is an acronym for Asynchronous JavaScript and XML. It is a group of inter-related technologies like JavaScript, DOM, XML, HTML/XHTML, CSS, XMLHttpRequest etc.

AJAX allows you to send and receive data asynchronously without reloading the web page. So it is fast.

AJAX allows you to send only important information to the server not the entire page. So only valuable data from the client side is routed to the server side. It makes your application interactive and faster.

Where it is used?

There are too many web applications running on the web that are using ajax technology like gmail, facebook,twitter, google map, youtube etc.

Ajax bootstrap :

Ajax is used to communicate with web pages and web servers. Some of examples which are based on ajax and bootstrap as shown below −

Example Description Download link

Payment Form This example indicates about Ajax Payment Form in Bootstrap

AJAX / DIV Wizard This example indicates about Ajax AJAX / DIV Wizard in Bootstrap

Bootstrap Navigation Components:

Tabs and Pills

Class Description Example

.nav nav-tabs Creates navigation tabs

.nav nav-pills Creates navigation pills

.nav nav-pills nav-stacked Creates vertical navigation pills

.nav-justified Makes navigation tabs/pills equal widths of their parent, at screens wider than 768px. On smaller screens, the nav tabs/pills are stacked

.disabled Indicates a disabled (unclickable) tab/pill

Navigation tabs with dropdown menu

Navigation pills with dropdown menu

.tab-content Together with .tab-pane and data-toggle="tab" (data-toggle="pill" for pills), it makes the tab/pill toggleable

.tab-pane Together with .tab-content and data-toggle="tab" (data-toggle="pill" for pills), it makes the tab/pill toggleable

Navbars

Class Description Example

.navbar Creates a navigation bar

.navbar-brand Added to a link or a header element inside the navbar to represent a logo or a header

.navbar-btn Vertically aligns a button inside a navbar

.navbar-collapse Collapses the navbar (hidden and replaced with a menu/hamburger icon on mobile phones and small tablets)

.navbar-default Creates a default navigation bar (light-grey background color)

.navbar-fixed-bottom Makes the navbar stay at the bottom of the screen (sticky/fixed)

.navbar-fixed-top Makes the navbar stay at the top of the screen (sticky/fixed)

.navbar-form Added to form elements inside the navbar to vertically center them (proper padding)

.navbar-header Added to a container element that contains the link/element that represent a logo or a header

.navbar-inverse Creates a black navigation bar (instead of light-grey)

.navbar-left Aligns nav links, forms, buttons, or text, in the navbar to the left

.navbar-link Styles an element to look like a link inside the navbar (anchors get proper padding and an underline on hover, while other elements like p or span gets a default hover effect - white color in an inversed navbar and a black color in a default navbar)

.navbar-nav Used on a <ul> container that contains the list items with links inside a navigation bar

.navbar-right Aligns nav links, forms, buttons, or text in the navbar to the right.

.navbar-static-top Removes left, top and right borders (rounded corners) from the navbar (default navbar has a gray border and a 4px border-radius by default)

.navbar-text Vertical align any elements inside the navbar that are not links (ensures proper padding)

.navbar-toggle Styles the button that should open the navbar on small screens. Often used together with three .icon-bar classes to indicate a toggleable menu icon (hamburger/bars)