Chapter 8:
Object-oriented Programming

In the early nineties, a thing called object-oriented programming stirred up the software industry. Most of the ideas behind it were not really new at the time, but they had finally gained enough momentum to start rolling, to become fashionable. Books were being written, courses given, programming languages developed. All of a sudden, everybody was extolling the virtues of object-orientation, enthusiastically applying it to every problem, convincing themselves they had finally found the right way to write programs.

These things happen a lot. When a process is hard and confusing, people are always on the lookout for a magic solution. When something looking like such a solution presents itself, they are prepared to become devoted followers. For many programmers, even today, object-orientation (or their view of it) is the gospel. When a program is not 'truly object-oriented', whatever that means, it is considered decidedly inferior.

Few fads have managed to stay popular for as long as this one, though. Object-orientation's longevity can largely be explained by the fact that the ideas at its core are very solid and useful. In this chapter, we will discuss these ideas, along with JavaScript's (rather eccentric) take on them. The above paragraphs are by no means meant to discredit these ideas. What I want to do is warn the reader against developing an unhealthy attachment to them.

As the name suggests, object-oriented programming is related to objects. So far, we have used objects as loose aggregations of values, adding and altering their properties whenever we saw fit. In an object-oriented approach, objects are viewed as little worlds of their own, and the outside world may touch them only through a limited and well-defined interface, a number of specific methods and properties. The 'reached list' we used at the end of chapter 7 is an example of this: We used only three functions, `makeReachedList`, `storeReached`, and `findReached` to interact with it. These three functions form an interface for such objects.

The `Date`, `Error`, and `BinaryHeap` objects we have seen also work like this. Instead of providing regular functions for working with the objects, they provide a way to create such objects, using the `new` keyword, and a number of methods and properties that provide the rest of the interface.

One way to give an object methods is to simply attach function values to it.

```
var rabbit = {};
rabbit.speak = function(line) {
  print("The rabbit says '", line, "'");
};

rabbit.speak("Well, now you're asking me.");
```

In most cases, the method will need to know who it should act on. For example, if there are different rabbits, the `speak` method must indicate which rabbit is speaking. For this purpose, there is a special variable called `this`, which is always present when a function is called, and which points at the relevant object when the function is called as a method. A function is called as a method when it is looked up as a property, and immediately called, as in `object.method()`.

```
function speak(line) {
  print("The ", this.adjective, " rabbit says '", line, "'");
}
var whiteRabbit = {adjective: "white", speak: speak};
var fatRabbit = {adjective: "fat", speak: speak};

whiteRabbit.speak("Oh my ears and whiskers, how late it's getting!");
fatRabbit.speak("I could sure use a carrot right now.");
```

I can now clarify the mysterious first argument to the `apply` method, for which we always used `null` in chapter 6. This argument can be used to specify the object that the function must be applied to. For non-method functions, this is irrelevant, hence the `null`.

```
speak.apply(fatRabbit, ["Yum."]);
```

Functions also have a `call` method, which is similar to `apply`, but you can give the arguments for the function separately instead of as an array:

```
speak.call(fatRabbit, "Burp.");
```

The `new` keyword provides a convenient way of creating new objects. When a function is called with the word `new` in front of it, its `this` variable will point at a new object, which it will automatically return (unless it explicitly returns something else). Functions used to create new objects like this are called constructors. Here is a constructor for rabbits:

```
function Rabbit(adjective) {
  this.adjective = adjective;
  this.speak = function(line) {
    print("The ", this.adjective, " rabbit says '", line, "'");
  };
}

var killerRabbit = new Rabbit("killer");
killerRabbit.speak("GRAAAAAAAAAH!");
```

It is a convention, among JavaScript programmers, to start the names of constructors with a capital letter. This makes it easy to distinguish them from other functions.

Why is the `new` keyword even necessary? After all, we could have simply written this:

```
function makeRabbit(adjective) {
  return {
    adjective: adjective,
    speak: function(line) {/*etc*/}
  };
}

var blackRabbit = makeRabbit("black");
```

But that is not entirely the same. `new` does a few things behind the scenes. For one thing, our `killerRabbit` has a property called `constructor`, which points at the `Rabbit` function that created it. `blackRabbit` also has such a property, but it points at the `Object` function.

```
show(killerRabbit.constructor);
show(blackRabbit.constructor);
```

Where did the `constructor` property come from? It is part of the prototype of a rabbit. Prototypes are a powerful, if somewhat confusing, part of the way JavaScript objects work. Every object is based on a prototype, which gives it a set of inherent properties. The simple objects we have used so far are based on the most basic prototype, which is associated with the `Object` constructor. In fact, typing `{}` is equivalent to typing `new Object()`.

```
var simpleObject = {};
show(simpleObject.constructor);
show(simpleObject.toString);
```

`toString` is a method that is part of the `Object` prototype. This means that all simple objects have a `toString` method, which converts them to a string. Our rabbit objects are based on the prototype associated with the `Rabbit` constructor. You can use a constructor's `prototype` property to get access to, well, their prototype:

```
show(Rabbit.prototype);
show(Rabbit.prototype.constructor);
```

Every function automatically gets a `prototype` property, whose `constructor` property points back at the function. Because the rabbit prototype is itself an object, it is based on the `Object` prototype, and shares its `toString` method.

```
show(killerRabbit.toString == simpleObject.toString);
```

Even though objects seem to share the properties of their prototype, this sharing is one-way. The properties of the prototype influence the object based on it, but the properties of this object never change the prototype.

The precise rules are this: When looking up the value of a property, JavaScript first looks at the properties that the object itself has. If there is a property that has the name we are looking for, that is the value we get. If there is no such property, it continues searching the prototype of the object, and then the prototype of the prototype, and so on. If no property is found, the value `undefined` is given. On the other hand, when setting the value of a property, JavaScript never goes to the prototype, but always sets the property in the object itself.

```
Rabbit.prototype.teeth = "small";
show(killerRabbit.teeth);
killerRabbit.teeth = "long, sharp, and bloody";
show(killerRabbit.teeth);
show(Rabbit.prototype.teeth);
```

This does mean that the prototype can be used at any time to add new properties and methods to all objects based on it. For example, it might become necessary for our rabbits to dance.

```
Rabbit.prototype.dance = function() {
  print("The ", this.adjective, " rabbit dances a jig.");
};

killerRabbit.dance();
```

And, as you might have guessed, the prototypical rabbit is the perfect place for values that all rabbits have in common, such as the `speak` method. Here is a new approach to the `Rabbit` constructor:

```
function Rabbit(adjective) {
  this.adjective = adjective;
}
Rabbit.prototype.speak = function(line) {
  print("The ", this.adjective, " rabbit says '", line, "'");
};

var hazelRabbit = new Rabbit("hazel");
hazelRabbit.speak("Good Frith!");
```

The fact that all objects have a prototype and receive some properties from this prototype can be tricky. It means that using an object to store a set of things, such as the cats from chapter 4, can go wrong. If, for example, we wondered whether there is a cat called `"constructor"`, we would have checked it like this:

```
var noCatsAtAll = {};
if ("constructor" in noCatsAtAll)
  print("Yes, there definitely is a cat called 'constructor'.");
```

This is problematic. A related problem is that it can often be practical to extend the prototypes of standard constructors such as `Object` and `Array` with new useful functions. For example, we could give all objects a method called `properties`, which returns an array with the names of the (non-hidden) properties that the object has:

```
Object.prototype.properties = function() {
  var result = [];
  for (var property in this)
    result.push(property);
  return result;
};

var test = {x: 10, y: 3};
show(test.properties());
```

And that immediately shows the problem. Now that the `Object` prototype has a property called `properties`, looping over the properties of any object, using `for` and `in`, will also give us that shared property, which is generally not what we want. We are interested only in the properties that the object itself has.

Fortunately, there is a way to find out whether a property belongs to the object itself or to one of its prototypes. Unfortunately, it does make looping over the properties of an object a bit clumsier. Every object has a method called `hasOwnProperty`, which tells us whether the object has a property with a given name. Using this, we could rewrite our `properties` method like this:

```
Object.prototype.properties = function() {
  var result = [];
  for (var property in this) {
    if (this.hasOwnProperty(property))
      result.push(property);
  }
  return result;
};

var test = {"Fat Igor": true, "Fireball": true};
show(test.properties());
```

And of course, we can abstract that into a higher-order function. Note that the `action` function is called with both the name of the property and the value it has in the object.

```
function forEachIn(object, action) {
  for (var property in object) {
    if (object.hasOwnProperty(property))
      action(property, object[property]);
  }
}

var chimera = {head: "lion", body: "goat", tail: "snake"};
forEachIn(chimera, function(name, value) {
  print("The ", name, " of a ", value, ".");
});
```

But, what if we find a cat named `hasOwnProperty`? (You never know.) It will be stored in the object, and the next time we want to go over the collection of cats, calling `object.hasOwnProperty` will fail, because that property no longer points at a function value. This can be solved by doing something even uglier:

```
function forEachIn(object, action) {
  for (var property in object) {
    if (Object.prototype.hasOwnProperty.call(object, property))
      action(property, object[property]);
  }
}

var test = {name: "Mordecai", hasOwnProperty: "Uh-oh"};
forEachIn(test, function(name, value) {
  print("Property ", name, " = ", value);
});
```

(Note: This example does not currently work correctly in Internet Explorer 8, which apparently has some problems with overriding built-in prototype properties.)

Here, instead of using the method found in the object itself, we get the method from the `Object` prototype, and then use `call` to apply it to the right object. Unless someone actually messes with the method in `Object.prototype` (don't do that), this should work correctly.

`hasOwnProperty` can also be used in those situations where we have been using the `in` operator to see whether an object has a specific property. There is one more catch, however. We saw in chapter 4 that some properties, such as `toString`, are 'hidden', and do not show up when going over properties with `for/in`. It turns out that browsers in the Gecko family (Firefox, most importantly) give every object a hidden property named `__proto__`, which points to the prototype of that object. `hasOwnProperty` will return `true` for this one, even though the program did not explicitly add it. Having access to the prototype of an object can be very convenient, but making it a property like that was not a very good idea. Still, Firefox is a widely used browser, so when you write a program for the web you have to be careful with this. There is a method `propertyIsEnumerable`, which returns `false` for hidden properties, and which can be used to filter out strange things like `__proto__`. An expression such as this one can be used to reliably work around this:

```
var object = {foo: "bar"};
show(Object.prototype.hasOwnProperty.call(object, "foo") &&
     Object.prototype.propertyIsEnumerable.call(object, "foo"));
```

Nice and simple, no? This is one of the not-so-well-designed aspects of JavaScript. Objects play both the role of 'values with methods', for which prototypes work great, and 'sets of properties', for which prototypes only get in the way.

Writing the above expression every time you need to check whether a property is present in an object is unworkable. We could put it into a function, but an even better approach is to write a constructor and a

prototype specifically for situations like this, where we want to approach an object as just a set of properties. Because you can use it to look things up by name, we will call it a `Dictionary`.

```
function Dictionary(startValues) {
  this.values = startValues || {};
}
Dictionary.prototype.store = function(name, value) {
  this.values[name] = value;
};
Dictionary.prototype.lookup = function(name) {
  return this.values[name];
};
Dictionary.prototype.contains = function(name) {
  return Object.prototype.hasOwnProperty.call(this.values, name) &&
    Object.prototype.propertyIsEnumerable.call(this.values, name);
};
Dictionary.prototype.each = function(action) {
  forEachIn(this.values, action);
};

var colours = new Dictionary({Grover: "blue",
                              Elmo: "orange",
                              Bert: "yellow"});
show(colours.contains("Grover"));
show(colours.contains("constructor"));
colours.each(function(name, colour) {
  print(name, " is ", colour);
});
```

Now the whole mess related to approaching objects as plain sets of properties has been 'encapsulated' in a convenient interface: one constructor and four methods. Note that the `values` property of a `Dictionary` object is not part of this interface, it is an internal detail, and when you are using `Dictionary` objects you do not need to directly use it.

Whenever you write an interface, it is a good idea to add a comment with a quick sketch of what it does and how it should be used. This way, when someone, possibly yourself three months after you wrote it, wants to work with the interface, they can quickly see how to use it, and do not have to study the whole program.

Most of the time, when you are designing an interface, you will soon find some limitations and problems in whatever you came up with, and change it. To prevent wasting your time, it is advisable to document your interfaces only after they have been used in a few real situations and proven themselves to be practical. — Of course, this might make it tempting to forget about documentation altogether. Personally, I treat writing documentation as a 'finishing touch' to add to a system. When it feels ready, it is time to write something about it, and to see if it sounds as good in English (or whatever language) as it does in JavaScript (or whatever programming language).

The distinction between the external interface of an object and its internal details is important for two reasons. Firstly, having a small, clearly described interface makes an object easier to use. You only have to keep the interface in mind, and do not have to worry about the rest unless you are changing the object itself.

Secondly, it often turns out to be necessary or practical to change something about the internal implementation of an object type[1], to make it more efficient, for example, or to fix some problem. When outside code is accessing every single property and detail in the object, you can not change any of them without also updating a lot of other code. If outside code only uses a small interface, you can do what you want, as long as you do not change the interface.

Some people go very far in this. They will, for example, never include properties in the interface of object, only methods — if their object type has a length, it will be accessible with the `getLength` method, not

the `length` property. This way, if they ever want to change their object in such a way that it no longer has a `length` property, for example because it now has some internal array whose length it must return, they can update the function without changing the interface.

My own take is that in most cases this is not worth it. Adding a `getLength` method which only contains `return this.length;` mostly just adds meaningless code, and, in most situations, I consider meaningless code a bigger problem than the risk of having to occasionally change the interface to my objects.

Adding new methods to existing prototypes can be very convenient. Especially the `Array` and `String` prototypes in JavaScript could use a few more basic methods. We could, for example, replace `forEach` and `map` with methods on arrays, and make the `startsWith` function we wrote in chapter 4 a method on strings.

However, if your program has to run on the same web-page as another program (either written by you or by someone else) which uses `for`/`in` naively — the way we have been using it so far — then adding things to prototypes, especially the `Object` and `Array` prototype, will definitely break something, because these loops will suddenly start seeing those new properties. For this reason, some people prefer not to touch these prototypes at all. Of course, if you are careful, and you do not expect your code to have to coexist with badly-written code, adding methods to standard prototypes is a perfectly good technique.

In this chapter we are going to build a virtual terrarium, a tank with insects moving around in it. There will be some objects involved (this is, after all, the chapter on object-oriented programming). We will take a rather simple approach, and make the terrarium a two-dimensional grid, like the second map in chapter 7. On this grid there are a number of bugs. When the terrarium is active, all the bugs get a chance to take an action, such as moving, every half second.

Thus, we chop both time and space into units with a fixed size — squares for space, half seconds for time. This usually makes things easier to model in a program, but of course has the drawback of being wildly inaccurate. Fortunately, this terrarium-simulator is not required to be accurate in any way, so we can get away with it.

A terrarium can be defined with a 'plan', which is an array of strings. We could have used a single string, but because JavaScript strings must stay on a single line it would have been a lot harder to type.

```
var thePlan =
  ["############################",
   "#      #    #      o      ##",
   "#                          #",
   "#          #####           #",
   "##         #   #    ##     #",
   "###           ##     #     #",
   "#           ###      #     #",
   "#   ####                   #",
   "#   ##       o             #",
   "# o  #         o       ### #",
   "#    #                     #",
   "############################"];
```

The `"#"` characters are used to represent the walls of the terrarium (and the ornamental rocks lying in it), the `"o"`s represent bugs, and the spaces are, as you might have guessed, empty space.

Such a plan-array can be used to create a terrarium-object. This object keeps track of the shape and content of the terrarium, and lets the bugs inside move. It has four methods: Firstly `toString`, which converts the terrarium back to a string similar to the plan it was based on, so that you can see what is going on inside it. Then there is `step`, which allows all the bugs in the terrarium to move one step, if they so desire. And finally, there are `start` and `stop`, which control whether the terrarium is 'running'. When it is running, `step` is automatically called every half second, so the bugs keep moving.

Ex. 8.1 The points on the grid will be represented by objects again. In chapter 7 we used three functions, `point`, `addPoints`, and `samePoint` to work with points. This time, we will use a constructor and two methods. Write the constructor `Point`, which takes two arguments, the x and y coordinates of the point, and produces an object with `x` and `y` properties. Give the prototype of this constructor a method `add`, which takes another point as argument and returns a new point whose `x` and `y` are the sum of the `x` and `y` of the two given points. Also add a method `isEqualTo`, which takes a point and returns a boolean indicating whether the `this` point refers to the same coordinates as the given point.

Apart from the two methods, the `x` and `y` properties are also part of the interface of this type of objects: Code which uses point objects may freely retrieve and modify `x` and `y`.

[show solution]

When writing objects to implement a certain program, it is not always very clear which functionality goes where. Some things are best written as methods of your objects, other things are better expressed as separate functions, and some things are best implemented by adding a new type of object. To keep things clear and organised, it is important to keep the amount of methods and responsibilities that an object type has as small as possible. When an object does too much, it becomes a big mess of functionality, and a formidable source of confusion.

I said above that the terrarium object will be responsible for storing its contents and for letting the bugs inside it move. Firstly, note that it lets them move, it doesn't make them move. The bugs themselves will also be objects, and these objects are responsible for deciding what they want to do. The terrarium merely provides the infrastructure that asks them what to do every half second, and if they decide to move, it makes sure this happens.

Storing the grid on which the content of the terrarium is kept can get quite complex. It has to define some kind of representation, ways to access this representation, a way to initialise the grid from a 'plan' array, a way to write the content of the grid to a string for the `toString` method, and the movement of the bugs on the grid. It would be nice if part of this could be moved into another object, so that the terrarium object itself doesn't get too big and complex.

Whenever you find yourself about to mix data representation and problem-specific code in one object, it is a good idea to try and put the data representation code into a separate type of object. In this case, we need to represent a grid of values, so I wrote a `Grid` type, which supports the operations that the terrarium will need.

To store the values on the grid, there are two options. One can use an array of arrays, like this:

```
var grid = [["0,0", "1,0", "2,0"],
            ["0,1", "1,1", "2,1"]];
show(grid[1][2]);
```

Or the values can all be put into a single array. In this case, the element at `x`,`y` can be found by getting the element at position `x + y * width` in the array, where `width` is the width of the grid.

```
var grid = ["0,0", "1,0", "2,0",
            "0,1", "1,1", "2,1"];
show(grid[2 + 1 * 3]);
```

I chose the second representation, because it makes it much easier to initialise the array. `new Array(x)` produces a new array of length `x`, filled with `undefined` values.

```
function Grid(width, height) {
  this.width = width;
  this.height = height;
  this.cells = new Array(width * height);
}
Grid.prototype.valueAt = function(point) {
  return this.cells[point.y * this.width + point.x];
};
Grid.prototype.setValueAt = function(point, value) {
  this.cells[point.y * this.width + point.x] = value;
};
Grid.prototype.isInside = function(point) {
  return point.x >= 0 && point.y >= 0 &&
         point.x < this.width && point.y < this.height;
};
Grid.prototype.moveValue = function(from, to) {
  this.setValueAt(to, this.valueAt(from));
  this.setValueAt(from, undefined);
};
```

Ex. 8.2 We will also need to go over all the elements of the grid, to find the bugs we need to move, or to convert the whole thing to a string. To make this easy, we can use a higher-order function that takes an action as its argument. Add the method `each` to the prototype of `Grid`, which takes a function of two arguments as its argument. It calls this function for every point on the grid, giving it the point object for that point as its first argument, and the value that is on the grid at that point as second argument.

Go over the points starting at `0`,`0`, one row at a time, so that `1`,`0` is handled before `0`,`1`. This will make it easier to write the `toString` function of the terrarium later. (Hint: Put a `for` loop for the `x` coordinate inside a loop for the `y` coordinate.)

It is advisable not to muck about in the `cells` property of the grid object directly, but use `valueAt` to get at the values. This way, if we decide (for some reason) to use a different method for storing the values, we only have to rewrite `valueAt` and `setValueAt`, and the other methods can stay untouched.

[show solution]

Finally, to test the grid:

```
var testGrid = new Grid(3, 2);
testGrid.setValueAt(new Point(1, 0), "#");
testGrid.setValueAt(new Point(1, 1), "o");
testGrid.each(function(point, value) {
  print(point.x, ",", point.y, ": ", value);
});
```

Before we can start to write a `Terrarium` constructor, we will have to get a bit more specific about these 'bug objects' that will be living inside it. Earlier, I mentioned that the terrarium will ask the bugs what action they want to take. This will work as follows: Each bug object has an `act` method which, when called, returns an 'action'. An action is an object with a `type` property, which names the type of action the bug wants to take, for example `"move"`. For most actions, the action also contains extra information, such as the direction the bug wants to go.

Bugs are terribly myopic, they can only see the squares directly around them on the grid. But these they can use to base their action on. When the `act` method is called, it is given an object with information about the surroundings of the bug in question. For each of the eight directions, it contains a property. The property indicating what is above the bug is called `"n"`, for North, the one indicating what is above and to the right `"ne"`, for North-East, and so on. To look up the direction these names refer to, the following dictionary object is useful:

```
var directions = new Dictionary(
  {"n":  new Point( 0, -1),
   "ne": new Point( 1, -1),
   "e":  new Point( 1,  0),
   "se": new Point( 1,  1),
   "s":  new Point( 0,  1),
   "sw": new Point(-1,  1),
   "w":  new Point(-1,  0),
   "nw": new Point(-1, -1)});

show(new Point(4, 4).add(directions.lookup("se")));
```

When a bug decides to move, he indicates in which direction he wants to go by giving the resulting action object a `direction` property that names one of these directions. We can make a simple, stupid bug that always just goes south, 'towards the light', like this:

```
function StupidBug() {};
StupidBug.prototype.act = function(surroundings) {
  return {type: "move", direction: "s"};
};
```

Now we can start on the `Terrarium` object type itself. First, its constructor, which takes a plan (an array of strings) as argument, and initialises its grid.

```
var wall = {};

function Terrarium(plan) {
  var grid = new Grid(plan[0].length, plan.length);
  for (var y = 0; y < plan.length; y++) {
    var line = plan[y];
    for (var x = 0; x < line.length; x++) {
      grid.setValueAt(new Point(x, y),
                      elementFromCharacter(line.charAt(x)));
    }
  }
  this.grid = grid;
}

function elementFromCharacter(character) {
  if (character == " ")
    return undefined;
  else if (character == "#")
    return wall;
  else if (character == "o")
    return new StupidBug();
}
```

`wall` is an object that is used to mark the location of walls on the grid. Like a real wall, it doesn't do much, it just sits there and takes up space.

The most straightforward method of a terrarium object is `toString`, which transforms a terrarium into a string. To make this easier, we mark both the `wall` and the prototype of the `StupidBug` with a property `character`, which holds the character that represents them.

```
wall.character = "#";
StupidBug.prototype.character = "o";

function characterFromElement(element) {
  if (element == undefined)
    return " ";
  else
    return element.character;
}

show(characterFromElement(wall));
```

Ex. 8.3 Now we can use the `each` method of the `Grid` object to build up a string. But to make the result readable, it would be nice to have a newline at the end of every row. The `x` coordinate of the positions on the grid can be used to determine when the end of a line is reached. Add a method `toString` to the `Terrarium` prototype, which takes no arguments and returns a string that, when given to `print`, shows a nice two-dimensional view of the terrarium.

[show solution]

It is possible that, when trying to solve the above exercise, you have tried to access `this.grid` inside the function that you pass as an argument to the grid's `each` method. This will not work. Calling a function always results in a new `this` being defined inside that function, even when it is not used as a method. Thus, any `this` variable outside of the function will not be visible.

Sometimes it is straightforward to work around this by storing the information you need in a variable, like `endOfLine`, which is visible in the inner function. If you need access to the whole `this` object, you can store that in a variable too. The name `self` (or `that`) is often used for such a variable.

But all these extra variables can get messy. Another good solution is to use a function similar to `partial` from chapter 6. Instead of adding arguments to a function, this one adds a `this` object, using the first argument to the function's `apply` method:

```
function bind(func, object) {
  return function(){
    return func.apply(object, arguments);
  };
}

var testArray = [];
var pushTest = bind(testArray.push, testArray);
pushTest("A");
pushTest("B");
show(testArray);
```

This way, you can `bind` an inner function to `this`, and it will have the same `this` as the outer function.

Ex. 8.4 In the expression `bind(testArray.push, testArray)` the name `testArray` still occurs twice. Can you design a function `method`, which allows you to bind an object to one of its methods without naming the object twice?

[show solution]

We will need `bind` (or `method`) when implementing the `step` method of a terrarium. This method has to go over all the bugs on the grid, ask them for an action, and execute the given action. You might be tempted to use `each` on the grid, and just handle the bugs we come across. But then, when a bug moves South or East, we will come across it again in the same turn, and allow it to move again.

Instead, we first gather all the bugs into an array, and then process them. This method gathers bugs, or other things that have an `act` method, and stores them in objects that also contain their current position:

```
Terrarium.prototype.listActingCreatures = function() {
  var found = [];
  this.grid.each(function(point, value) {
    if (value != undefined && value.act)
      found.push({object: value, point: point});
  });
  return found;
};
```

Ex. 8.5 When asking a bug to act, we must pass it an object with information about its current surroundings. This object will use the direction names we saw earlier (`"n"`, `"ne"`, etcetera) as property names. Each property holds a string of one character, as returned by `characterFromElement`, indicating what the bug can see in that direction.

Add a method `listSurroundings` to the `Terrarium` prototype. It takes one argument, the point at which the bug is currently standing, and returns an object with information about the surroundings of that point. When the point is at the edge of the grid, use `"#"` for the directions that go outside of the grid, so the bug will not try to move there.

Hint: Do not write out all the directions, use the `each` method on the `directions` dictionary.

[show solution]

Both above methods are not part of the external interface of a `Terrarium` object, they are internal details. Some languages provide ways to explicitly declare certain methods and properties 'private', and make it an error to use them from outside the object. JavaScript does not, so you will have to rely on comments to describe the interface to an object. Sometimes it can be useful to use some kind of naming scheme to distinguish between external and internal properties, for example by prefixing all internal ones with an underscore ('_'). This will make accidental uses of properties that are not part of an object's interface easier to spot.

Next is one more internal method, the one that will ask a bug for an action and carry it out. It takes an object with `object` and `point` properties, as returned by `listActingCreatures`, as argument. For now, it only knows about the `"move"` action:

```
Terrarium.prototype.processCreature = function(creature) {
  var surroundings = this.listSurroundings(creature.point);
  var action = creature.object.act(surroundings);
  if (action.type == "move" && directions.contains(action.direction)) {
    var to = creature.point.add(directions.lookup(action.direction));
    if (this.grid.isInside(to) && this.grid.valueAt(to) == undefined)
      this.grid.moveValue(creature.point, to);
  }
  else {
    throw new Error("Unsupported action: " + action.type);
  }
};
```

Note that it checks whether the chosen direction is inside of the grid and empty, and ignores it otherwise. This way, the bugs can ask for any action they like — the action will only be carried out if it is actually possible. This acts as a layer of insulation between the bugs and the terrarium, and allows us to be less precise when writing the bugs' `act` methods — for example the `StupidBug` just always travels South, regardless of any walls that might stand in its way.

These three internal methods then finally allow us to write the `step` method, which gives all bugs a chance to do something (all elements with an `act` method — we could also give the `wall` object one if we so desired, and make the walls walk).

```
Terrarium.prototype.step = function() {
  forEach(this.listActingCreatures(),
          bind(this.processCreature, this));
};
```

Now, let us make a terrarium and see whether the bugs move...

```
var terrarium = new Terrarium(thePlan);
print(terrarium);
terrarium.step();
print(terrarium);
```

Wait, how come the above calls `print(terrarium)` and ends up displaying the output of our `toString` method? `print` turns its arguments to strings using the `String` function. Objects are turned to strings by calling their `toString` method, so giving your own object types a meaningful `toString` is a good way to make them readable when printed out.

```
Point.prototype.toString = function() {
  return "(" + this.x + "," + this.y + ")";
};
print(new Point(5, 5));
```

As promised, `Terrarium` objects also get `start` and `stop` methods to start or stop their simulation. For this, we will use two functions provided by the browser, called `setInterval` and `clearInterval`. The first is used to cause its first argument (a function, or a string containing JavaScript code) to be executed periodically. Its second argument gives the amount of milliseconds (1/1000 second) between invocations. It returns a value that can be given to `clearInterval` to stop its effect.

```
var annoy = setInterval(function() {print("What?");}, 400);
```

And...

```
clearInterval(annoy);
```

There are similar functions for one-shot time-based actions. `setTimeout` causes a function or string to be executed after a given amount of milliseconds, and `clearTimeout` cancels such an action.

```
Terrarium.prototype.start = function() {
  if (!this.running)
    this.running = setInterval(bind(this.step, this), 500);
};

Terrarium.prototype.stop = function() {
  if (this.running) {
    clearInterval(this.running);
    this.running = null;
  }
};
```

Now we have a terrarium with some simple-minded bugs, and we can run it. But to see what is going on, we have to repeatedly do `print(terrarium)`, or we won't see what is going on. That is not very practical. It would be nicer if it would print automatically. It would also look better if, instead of printing a

thousand terraria below each other, we could update a single printout of the terrarium. For that second problem, this page conveniently provides a function called `inPlacePrinter`. It returns a function like `print` which, instead of adding to the output, replaces its previous output.

```
var printHere = inPlacePrinter();
printHere("Now you see it.");
setTimeout(partial(printHere, "Now you don't."), 1000);
```

To cause the terrarium to be re-printed every time it changes, we can modify the `step` method as follows:

```
Terrarium.prototype.step = function() {
  forEach(this.listActingCreatures(),
          bind(this.processCreature, this));
  if (this.onStep)
    this.onStep();
};
```

Now, when an `onStep` property has been added to a terrarium, it will be called on every step.

```
var terrarium = new Terrarium(thePlan);
terrarium.onStep = partial(inPlacePrinter(), terrarium);
terrarium.start();
```

Note the use of `partial` — it produces an in-place printer applied to the terrarium. Such a printer only takes one argument, so after partially applying it there are no arguments left, and it becomes a function of zero arguments. That is exactly what we need for the `onStep` property.

Don't forget to stop the terrarium when it is no longer interesting (which should be pretty soon), so that it does not keep wasting your computer's resources:

```
terrarium.stop();
```

But who wants a terrarium with just one kind of bug, and a stupid bug at that? Not me. It would be nice if we could add different kinds of bugs. Fortunately, all we have to do is to make the `elementFromCharacter` function more general. Right now it contains three cases which are typed in directly, or 'hard-coded':

```
function elementFromCharacter(character) {
  if (character == " ")
    return undefined;
  else if (character == "#")
    return wall;
  else if (character == "o")
    return new StupidBug();
}
```

The first two cases we can leave intact, but the last one is way too specific. A better approach would be to store the characters and the corresponding bug-constructors in a dictionary, and look for them there:

```
var creatureTypes = new Dictionary();
creatureTypes.register = function(constructor) {
  this.store(constructor.prototype.character, constructor);
};

function elementFromCharacter(character) {
  if (character == " ")
    return undefined;
  else if (character == "#")
    return wall;
  else if (creatureTypes.contains(character))
    return new (creatureTypes.lookup(character))();
  else
    throw new Error("Unknown character: " + character);
}
```

Note how the `register` method is added to `creatureTypes` — this is a dictionary object, but there is no reason why it shouldn't support an additional method. This method looks up the character associated with a constructor, and stores it in the dictionary. It should only be called on constructors whose prototype does actually have a `character` property.

`elementFromCharacter` now looks up the character it is given in `creatureTypes`, and raises an exception when it comes across an unknown character.

Here is a new bug type, and the call to register its character in `creatureTypes`:

```
function BouncingBug() {
  this.direction = "ne";
}
BouncingBug.prototype.act = function(surroundings) {
  if (surroundings[this.direction] != " ")
    this.direction = (this.direction == "ne" ? "sw" : "ne");
  return {type: "move", direction: this.direction};
};
BouncingBug.prototype.character = "%";

creatureTypes.register(BouncingBug);
```

Can you figure out what it does?

Ex. 8.6 Create a bug type called `DrunkBug` which tries to move in a random direction every turn, never mind whether there is a wall there. Remember the `Math.random` trick from chapter 7.

[show solution]

So, let us test out our new bugs:

```
var newPlan =
  ["############################",
   "#                      #####",
   "#    ##                ####",
   "#   ####     ~ ~          ##",
   "#    ##       ~            #",
   "#                           #",
   "#                ###        #",
   "#               #####       #",
   "#                ###        #",
   "# %        ###        %     #",
   "#         #######           #",
   "############################"];

var terrarium = new Terrarium(newPlan);
terrarium.onStep = partial(inPlacePrinter(), terrarium);
terrarium.start();
```

Notice the bouncing bugs bouncing off the drunk ones? Pure drama. Anyway, when you are done watching this fascinating show, shut it down:

```
terrarium.stop();
```

We now have two kinds of objects that both have an `act` method and a `character` property. Because they share these traits, the terrarium can approach them in the same way. This allows us to have all kinds of bugs, without changing anything about the terrarium code. This technique is called polymorphism, and it is arguably the most powerful aspect of object-oriented programming.

The basic idea of polymorphism is that when a piece of code is written to work with objects that have a certain interface, any kind of object that happens to support this interface can be plugged into the code, and it will just work. We already saw simple examples of this, like the `toString` method on objects. All objects that have a meaningful `toString` method can be given to `print` and other functions that need to convert values to strings, and the correct string will be produced, no matter how their `toString` method chooses to build this string.

Similarly, `forEach` works on both real arrays and the pseudo-arrays found in the `arguments` variable, because all it needs is a `length` property and properties called `0`, `1`, and so on, for the elements of the array.

To make life in the terrarium more life-like, we will add to it the concepts of food and reproduction. Each living thing in the terrarium gets a new property, `energy`, which is reduced by performing actions, and increased by eating things. When it has enough energy, a thing can reproduce[2], generating a new creature of the same kind.

If there are only bugs, wasting energy by moving around and eating each other, a terrarium will soon succumb to the forces of entropy, run out of energy, and become a lifeless wasteland. To prevent this from happening (too quickly, at least), we add lichen to the terrarium. Lichen do not move, they just use photo-synthesis to gather energy, and reproduce.

To make this work, we will need a terrarium with a different `processCreature` method. We could just replace the method of to the `Terrarium` prototype, but we have become very attached to the simulation of the bouncing and drunk bugs, and we would hate to break our old terrarium.

What we can do is create a new constructor, `LifeLikeTerrarium`, whose prototype is based on the `Terrarium` prototype, but which has a different `processCreature` method.

There are a few ways to do this. We could go over the properties of `Terrarium.prototype`, and add them one by one to `LifeLikeTerrarium.prototype`. This is easy to do, and in some cases it is the

best solution, but in this case there is a cleaner way. If we make the old prototype object the prototype of the new prototype object (you may have to re-read that a few times), it will automatically have all its properties.

Unfortunately, JavaScript does not have a straightforward way to create an object whose prototype is a certain other object. It is possible to write a function that does this, though, by using the following trick:

```
function clone(object) {
  function OneShotConstructor(){}
  OneShotConstructor.prototype = object;
  return new OneShotConstructor();
}
```

This function uses an empty one-shot constructor, whose prototype is the given object. When using `new` on this constructor, it will create a new object based on the given object.

```
function LifeLikeTerrarium(plan) {
  Terrarium.call(this, plan);
}
LifeLikeTerrarium.prototype = clone(Terrarium.prototype);
LifeLikeTerrarium.prototype.constructor = LifeLikeTerrarium;
```

The new constructor doesn't need to do anything different from the old one, so it just calls the old one on the `this` object. We also have to restore the `constructor` property in the new prototype, or it would claim its constructor is `Terrarium` (which, of course, is only really a problem when we make use of this property, which we don't).

It is now possible to replace some of the methods of the `LifeLikeTerrarium` object, or add new ones. We have based a new object type on an old one, which saved us the work of re-writing all the methods which are the same in `Terrarium` and `LifeLikeTerrarium`. This technique is called 'inheritance'. The new type inherits the properties of the old type. In most cases, this means the new type will still support the interface of the old type, though it might also support a few methods that the old type does not have. This way, objects of the new type can be (polymorphically) used in all the places where objects of the old type could be used.

In most programming languages with explicit support for object-oriented programming, inheritance is a very straightforward thing. In JavaScript, the language doesn't really specify a simple way to do it. Because of this, JavaScript programmers have invented many different approaches to inheritance. Unfortunately, none of them is quite perfect. Fortunately, such a broad range of approaches allows a programmer to choose the most suitable one for the problem he is solving, and allows certain tricks that would be utterly impossible in other languages.

At the end of this chapter, I will show a few other ways to do inheritance, and the issues they have.

Here is the new `processCreature` method. It is big.

```
LifeLikeTerrarium.prototype.processCreature = function(creature) {
  if (creature.object.energy <= 0) return;
  var surroundings = this.listSurroundings(creature.point);
  var action = creature.object.act(surroundings);

  var target = undefined;
  var valueAtTarget = undefined;
  if (action.direction && directions.contains(action.direction)) {
    var direction = directions.lookup(action.direction);
    var maybe = creature.point.add(direction);
    if (this.grid.isInside(maybe)) {
      target = maybe;
      valueAtTarget = this.grid.valueAt(target);
    }
  }

  if (action.type == "move") {
    if (target && !valueAtTarget) {
      this.grid.moveValue(creature.point, target);
      creature.point = target;
      creature.object.energy -= 1;
    }
  }
  else if (action.type == "eat") {
    if (valueAtTarget && valueAtTarget.energy) {
      this.grid.setValueAt(target, undefined);
      creature.object.energy += valueAtTarget.energy;
      valueAtTarget.energy = 0;
    }
  }
  else if (action.type == "photosynthese") {
    creature.object.energy += 1;
  }
  else if (action.type == "reproduce") {
    if (target && !valueAtTarget) {
      var species = characterFromElement(creature.object);
      var baby = elementFromCharacter(species);
      creature.object.energy -= baby.energy * 2;
      if (creature.object.energy > 0)
        this.grid.setValueAt(target, baby);
    }
    else if (action.type == "wait") {
      creature.object.energy -= 0.2;
    }
    else {
      throw new Error("Unsupported action: " + action.type);
    }

    if (creature.object.energy <= 0)
      this.grid.setValueAt(creature.point, undefined);
};
```

The function still starts by asking the creature for an action, provided it isn't out of energy (dead). Then, if the action has a `direction` property, it immediately computes which point on the grid this direction points to and which value is currently sitting there. Three of the five supported actions need to know this, and the code would be even uglier if they all computed it separately. If there is no `direction` property, or an invalid one, it leaves the variables `target` and `valueAtTarget` undefined.

After this, it goes over all the actions. Some actions require additional checking before they are executed, this is done with a separate `if` so that if a creature, for example, tries to walk through a wall, we do not generate an `"Unsupported action"` exception.

Note that, in the `"reproduce"` action, the parent creature loses twice the energy that the newborn creature gets (childbearing is not easy), and the new creature is only placed on the grid if the parent had enough energy to produce it.

After the action has been performed, we check whether the creature is out of energy. If it is, it dies, and we remove it.

Lichen is not a very complex organism. We will use the character `"*"` to represent it. Make sure you have defined the `randomElement` function from exercise 8.6, because it is used again here.

```
function Lichen() {
  this.energy = 5;
}
Lichen.prototype.act = function(surroundings) {
  var emptySpace = findDirections(surroundings, " ");
  if (this.energy >= 13 && emptySpace.length > 0)
    return {type: "reproduce", direction: randomElement(emptySpace)};
  else if (this.energy < 20)
    return {type: "photosynthese"};
  else
    return {type: "wait"};
};
Lichen.prototype.character = "*";

creatureTypes.register(Lichen);

function findDirections(surroundings, wanted) {
  var found = [];
  directions.each(function(name) {
    if (surroundings[name] == wanted)
      found.push(name);
  });
  return found;
}
```

Lichen do not grow bigger than 20 energy, or they would get huge when they are surrounded by other lichen and have no room to reproduce.

Ex. 8.7 Create a `LichenEater` creature. It starts with an energy of `10`, and behaves in the following way:

- When it has an energy of 30 or more, and there is room near it, it reproduces.
- Otherwise, if there are lichen nearby, it eats a random one.
- Otherwise, if there is space to move, it moves into a random nearby empty square.
- Otherwise, it waits.

Use `findDirections` and `randomElement` to check the surroundings and to pick directions. Give the lichen-eater `"c"` as its character (pac-man).

[show solution]

And try it out.

```
var lichenPlan =
  ["############################",
   "#                      ######",
   "#    ***                **##",
   "#   *####          **  c  *##",
   "#    ***     c     ####    *#",
   "#       c          #####   *#",
   "#                  ####    *#",
   "#   c       #*             *#",
   "#*          #**       c    *#",
   "#***        ###*     c     **#",
   "#******     ######         *###",
   "############################"];

var terrarium = new LifeLikeTerrarium(lichenPlan);
terrarium.onStep = partial(inPlacePrinter(), terrarium);
terrarium.start();
```

Most likely, you will see the lichen quickly over-grow a large part of the terrarium, after which the abundance of food makes the eaters so numerous that they wipe out all the lichen, and thus themselves. Ah, tragedy of nature.

```
terrarium.stop();
```

Having the inhabitants of your terrarium go extinct after a few minutes is kind of depressing. To deal with this, we have to teach our lichen-eaters about long-term sustainable farming. By making them only eat if they see at least two lichen nearby, no matter how hungry they are, they will never exterminate the lichen. This requires some discipline, but the result is a biotope that does not destroy itself. Here is a new `act` method — the only change is that it now only eats when `lichen.length` is at least two.

```
LichenEater.prototype.act = function(surroundings) {
  var emptySpace = findDirections(surroundings, " ");
  var lichen = findDirections(surroundings, "*");

  if (this.energy >= 30 && emptySpace.length > 0)
    return {type: "reproduce", direction: randomElement(emptySpace)};
  else if (lichen.length > 1)
    return {type: "eat", direction: randomElement(lichen)};
  else if (emptySpace.length > 0)
    return {type: "move", direction: randomElement(emptySpace)};
  else
    return {type: "wait"};
};
```

Run the above `lichenPlan` terrarium again, and see how it goes. Unless you are very lucky, the lichen-eaters will probably still go extinct after a while, because, in a time of mass starvation, they crawl aimlessly back and forth through empty space, instead of finding the lichen that is sitting just around the corner.

Ex. 8.8 Find a way to modify the `LichenEater` to be more likely to survive. Do not cheat — `this.energy += 100` is cheating. If you rewrite the constructor, do not forget to re-register it in the `creatureTypes` dictionary, or the terrarium will continue to use the old constructor.

[show solution]

Ex. 8.9 A one-link food chain is still a bit rudimentary. Can you write a new creature, `LichenEaterEater` (character `"@"`), which survives by eating lichen-eaters? Try to find a way to make it fit in the ecosystem without dying out too quickly. Modify the `lichenPlan` array to include a few of these, and try them out.

[show solution]

That concludes our discussion of terraria. The rest of the chapter is devoted to a more in-depth look at inheritance, and the problems related to inheritance in JavaScript.

First, some theory. Students of object-oriented programming can often be heard having lengthy, subtle discussions about correct and incorrect uses of inheritance. It is important to bear in mind that inheritance, in the end, is just a trick that allows lazy[3] programmers to write less code. Thus, the question of whether inheritance is being used correctly boils down to the question of whether the resulting code works correctly and avoids useless repetitions. Still, the principles used by these students provide a good way to start thinking about inheritance.

Inheritance is the creation of a new type of objects, the 'sub-type', based on an existing type, the 'super-type'. The sub-type starts with all the properties and methods of the super-type, it inherits them, and then modifies a few of these, and optionally adds new ones. Inheritance is best used when the thing modelled by the sub-type can be said to be an object of the super-type.

Thus, a `Piano` type could be a sub-type of an `Instrument` type, because a piano is an instrument. Because a piano has a whole array of keys, one might be tempted to make `Piano` a sub-type of `Array`, but a piano is no array, and implementing it like that is bound to lead to all kinds of silliness. For example, a piano also has pedals. Why would `piano[0]` give me the first key, and not the first pedal? The situation is, of course, that a piano has keys, so it would be better to give it a property `keys`, and possibly another property `pedals`, both holding arrays.

It is possible for a sub-type to be the super-type of yet another sub-type. Some problems are best solved by building a complex family tree of types. You have to take care not to get too inheritance-happy, though. Overuse of inheritance is a great way to make a program into a big ugly mess.

The working of the `new` keyword and the `prototype` property of constructors suggest a certain way of using objects. For simple objects, such as the terrarium-creatures, this way works rather well. Unfortunately, when a program starts to make serious use of inheritance, this approach to objects quickly becomes clumsy. Adding some functions to take care of common operations can make things a little smoother. Many people define, for example, `inherit` and `method` methods on objects.

```
Object.prototype.inherit = function(baseConstructor) {
  this.prototype = clone(baseConstructor.prototype);
  this.prototype.constructor = this;
};
Object.prototype.method = function(name, func) {
  this.prototype[name] = func;
};

function StrangeArray(){}
StrangeArray.inherit(Array);
StrangeArray.method("push", function(value) {
  Array.prototype.push.call(this, value);
  Array.prototype.push.call(this, value);
});

var strange = new StrangeArray();
strange.push(4);
show(strange);
```

If you search the web for the words 'JavaScript' and 'inheritance', you will come across scores of different variations on this, some of them quite a lot more complex and clever than the above.

Note how the `push` method written here uses the `push` method from the prototype of its parent type. This is something that is done often when using inheritance — a method in the sub-type internally uses a method of the super-type, but extends it somehow.

The biggest problem with this basic approach is the duality between constructors and prototypes. Constructors take a very central role, they are the things that give an object type its name, and when you need to get at a prototype, you have to go to the constructor and take its `prototype` property.

Not only does this lead to a lot of typing (`"prototype"` is 9 letters), it is also confusing. We had to write an empty, useless constructor for `StrangeArray` in the example above. Quite a few times, I have found myself accidentally adding methods to a constructor instead of its prototype, or trying to call `Array.slice` when I really meant `Array.prototype.slice`. As far as I am concerned, the prototype itself is the most important aspect of an object type, and the constructor is just an extension of that, a special kind of method.

With a few simple helper methods added to `Object.prototype`, it is possible to create an alternative approach to objects and inheritance. In this approach, a type is represented by its prototype, and we will use capitalised variables to store these prototypes. When it needs to do any 'constructing' work, this is done by a method called `construct`. We add a method called `create` to the `Object` prototype, which is used in place of the `new` keyword. It clones the object, and calls its `construct` method, if there is such a method, giving it the arguments that were passed to `create`.

```
Object.prototype.create = function() {
  var object = clone(this);
  if (typeof object.construct == "function")
    object.construct.apply(object, arguments);
  return object;
};
```

Inheritance can be done by cloning a prototype object and adding or replacing some of its properties. We also provide a convenient shorthand for this, an `extend` method, which clones the object it is applied to and adds to this clone the properties in the object that it is given as an argument.

```
Object.prototype.extend = function(properties) {
  var result = clone(this);
  forEachIn(properties, function(name, value) {
    result[name] = value;
  });
  return result;
};
```

In a case where it is not safe to mess with the `Object` prototype, these can of course be implemented as regular (non-method) functions.

An example. If you are old enough, you may at one time have played a 'text adventure' game, where you move through a virtual world by typing commands, and get textual descriptions of the things around you and the actions you perform. Now those were games!

We could write the prototype for an item in such a game like this.

```
var Item = {
  construct: function(name) {
    this.name = name;
  },
  inspect: function() {
    print("it is ", this.name, ".");
  },
  kick: function() {
    print("klunk!");
  },
  take: function() {
    print("you can not lift ", this.name, ".");
  }
};

var lantern = Item.create("the brass lantern");
lantern.kick();
```

Inherit from it like this...

```
var DetailedItem = Item.extend({
  construct: function(name, details) {
    Item.construct.call(this, name);
    this.details = details;
  },
  inspect: function() {
    print("you see ", this.name, ", ", this.details, ".");
  }
});

var giantSloth = DetailedItem.create(
  "the giant sloth",
  "it is quietly hanging from a tree, munching leaves");
giantSloth.inspect();
```

Leaving out the compulsory `prototype` part makes things like calling `Item.construct` from `DetailedItem`'s constructor slightly simpler. Note that it would be a bad idea to just do `this.name = name` in `DetailedItem.construct`. This duplicates a line. Sure, duplicating the line is shorter than calling the `Item.construct` function, but if we end up adding something to this constructor later, we have to add it in two places.

Most of the time, a sub-type's constructor should start by calling the constructor of the super-type. This way, it starts with a valid object of the super-type, which it can then extend. In this new approach to prototypes, types that need no constructor can leave it out. They will automatically inherit the constructor of their super-type.

```
var SmallItem = Item.extend({
  kick: function() {
    print(this.name, " flies across the room.");
  },
  take: function() {
    // (imagine some code that moves the item to your pocket here)
    print("you take ", this.name, ".");
  }
});

var pencil = SmallItem.create("the red pencil");
pencil.take();
```

Even though `SmallItem` does not define its own constructor, creating it with a `name` argument works, because it inherited the constructor from the `Item` prototype.

JavaScript has an operator called `instanceof`, which can be used to determine whether an object is based on a certain prototype. You give it the object on the left hand side, and a constructor on the right hand side, and it returns a boolean, `true` if the constructor's `prototype` property is the direct or indirect prototype of the object, and `false` otherwise.

When you are not using regular constructors, using this operator becomes rather clumsy — it expects a constructor function as its second argument, but we only have prototypes. A trick similar to the `clone` function can be used to get around it: We use a 'fake constructor', and apply `instanceof` to it.

```
Object.prototype.hasPrototype = function(prototype) {
  function DummyConstructor() {}
  DummyConstructor.prototype = prototype;
  return this instanceof DummyConstructor;
};
```

```
show(pencil.hasPrototype(DetailedItem));
```

Next, we want to make a small item that has a detailed description. It seems like this item would have to inherit both from `DetailedItem` and `SmallItem`. JavaScript does not allow an object to have multiple prototypes, and even if it did, the problem would not be quite that easy to solve. For example, if `SmallItem` would, for some reason, also define an `inspect` method, which `inspect` method should the new prototype use?

Deriving an object type from more than one parent type is called multiple inheritance. Some languages chicken out and forbid it altogether, others define complicated schemes for making it work in a well-defined and practical way. It is possible to implement a decent multiple-inheritance framework in JavaScript. In fact there are, as usual, multiple good approaches to this. But they all are too complex to be discussed here. Instead, I will show a very simple approach which suffices in most cases.

A mix-in is a specific kind of prototype which can be 'mixed into' other prototypes. `SmallItem` can be seen as such a prototype. By copying its `kick` and `take` methods into another prototype, we mix smallness into this prototype.

```
function mixInto(object, mixIn) {
  forEachIn(mixIn, function(name, value) {
    object[name] = value;
  });
};

var SmallDetailedItem = clone(DetailedItem);
mixInto(SmallDetailedItem, SmallItem);

var deadMouse = SmallDetailedItem.create(
  "Fred the mouse",
  "he is dead");
deadMouse.inspect();
deadMouse.kick();
```

Remember that `forEachIn` only goes over the object's own properties, so it will copy `kick` and `take`, but not the constructor that `SmallItem` inherited from `Item`.

Mixing prototypes gets more complex when the mix-in has a constructor, or when some of its methods 'clash' with methods in the prototype that it is mixed into. Sometimes, it is workable to do a 'manual mix-

in'. Say we have a prototype `Monster`, which has its own constructor, and we want to mix that with `DetailedItem`.

```
var Monster = Item.extend({
  construct: function(name, dangerous) {
    Item.construct.call(this, name);
    this.dangerous = dangerous;
  },
  kick: function() {
    if (this.dangerous)
      print(this.name, " bites your head off.");
    else
      print(this.name, " runs away, weeping.");
  }
});

var DetailedMonster = DetailedItem.extend({
  construct: function(name, description, dangerous) {
    DetailedItem.construct.call(this, name, description);
    Monster.construct.call(this, name, dangerous);
  },
  kick: Monster.kick
});

var giantSloth = DetailedMonster.create(
  "the giant sloth",
  "it is quietly hanging from a tree, munching leaves",
  true);
giantSloth.kick();
```

But note that this causes `Item` constructor to be called twice when creating a `DetailedMonster` — once through the `DetailedItem` constructor, and once through the `Monster` constructor. In this case there is not much harm done, but there are situations where this would cause a problem.

But don't let those complications discourage you from making use of inheritance. Multiple inheritance, though extremely useful in some situations, can be safely ignored most of the time. This is why languages like Java get away with forbidding multiple inheritance. And if, at some point, you find that you really need it, you can search the web, do some research, and figure out an approach that works for your situation.

Now that I think about it, JavaScript would probably be a great environment for building a text adventure. The ability to change the behaviour of objects at will, which is what prototypical inheritance gives us, is very well suited for this. If you have an object `hedgehog`, which has the unique habit of rolling up when it is kicked, you can just change its `kick` method.

Unfortunately, the text adventure went the way of the vinyl record and, while once very popular, is nowadays only played by a small population of enthusiasts.

<< Previous chapter | Contents | Cover | Next chapter >>