**1 - VALUES, TYPES AND OPERATORS**

**Values** - numbers, strings, Booleans, objects, functions, and undefined values.

**Special Numbers -** There are three special values in JavaScript that are considered numbers but don’t behave like normal numbers. The first two are **Infinity** and **-Infinity**. Don’t put too much trust in infinity-based computation. It isn’t mathematically solid, and it will quickly lead to our next special number: **NaN**.

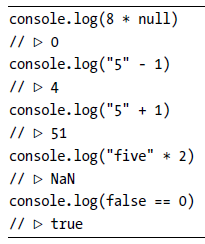
**Unary Operators -** console.log(**typeof** 4.5) // number; console.log(**typeof** "x") // string

**Booleans-** There is only one value in JavaScript that is not equal to itself, and that is NaN (“not a number”).

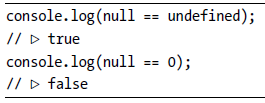
console.log(NaN == NaN) // *.* false

**Undefined Values - null** and **undefined** are used to denote the absence of a meaningful value. The difference in meaning between undefined and null is an accident of JavaScript’s design, and it doesn’t matter most of the time. I recommend treating them as interchangeable.

**Automatic Type Conversion**



When an operator is applied to the “wrong” type of value, JavaScript will quietly convert that value to the type it wants, using a set of rules that often aren’t what you want or expect. This is called ***type coercion***.

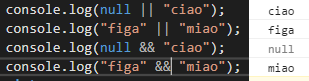


But what if you want to test whether something refers to the precise value false? The rules for converting strings and numbers to Boolean values state that 0, NaN, and the empty string ("") count as false, while all the other

values count as true.

If you do *not* want any automatic type conversions to happen, there are two extra operators: **===** and **!==**.

**Short-Circuiting of Logical Operators**



Expression to their right is evaluated only when necessary: This is called ***short-circuit evaluation***.

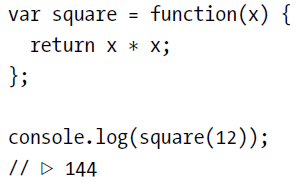
**2 - PROGRAM STRUCTURE**

**Expression**: code that produces a value.

**The Environment:** The collection of variables and their values that exist at a given time. When a program starts up, this environment is not empty.

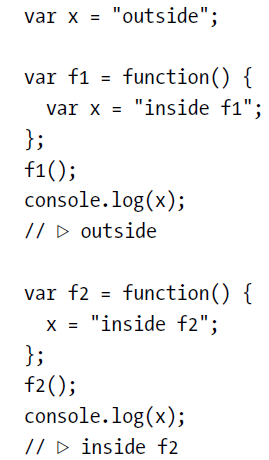
**3 – FUNCTIONS**

**Defining a Function:** variable definition where the value given to the variable happens to be a function.



The return keyword without an expression after it will cause the function to return undefined.

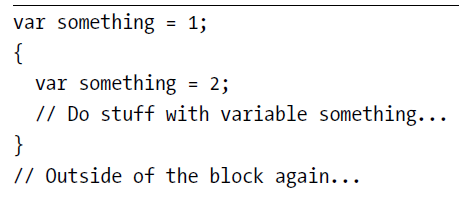
**Parameters and Scopes:** the variables created inside of functions, including their parameters, are *local* to the function. This “localness” of variables applies only to the parameters and to variables declared with the var keyword inside the function body. Variables declared outside of any function are called *global*, because they are visible throughout the program. It is possible to access such variables from inside a function, as long as you haven’t declared a local variable with the same name.



People who have experience with other programming languages might expect that any block of code between braces produces a new local environment. But in JavaScript, functions are the only things that create a new

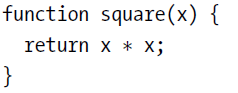
scope.

You are allowed to use free-standing blocks.

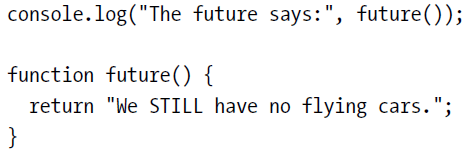


But the something inside the block refers to the same variable as the one outside the block.

**Declaration Notation:** The function keyword can also be used at the start of a statement, as in the following:



This is a function *declaration*. There is one subtlety with this form of function definition, however.



This code works, even though the function is defined *below* the code that uses it. This is because function declarations are not part of the regular top-to-bottom flow of control. They are conceptually moved to the top of

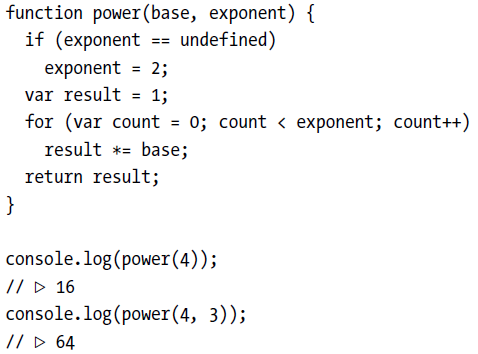
their scope and can be used by all the code in that scope. What happens when you put such a function definition inside a conditional (if) block or a loop? Well, don’t do that. Different JavaScript platforms in different browsers have traditionally done different things in that situation, and the latest standard actually forbids it. If you want your programs to behave consistently, only use this form of function-defining statements in the outermost block of a function or program.

**Optional arguments -** The function alert officially accepts only one argument.

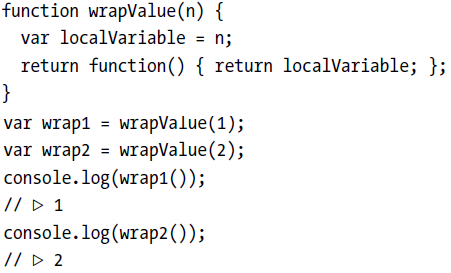


Yet when you call it like this, it doesn’t complain. It simply ignores the other arguments and shows you “Hello.” JavaScript is extremely broad-minded about the number of arguments you pass to a function. If you pass too many, the extra ones are ignored. If you pass too few, the missing parameters simply get assigned the value undefined. The downside of this is that it is possible that you’ll accidentally pass the wrong number of arguments to functions and no one will tell you about it.

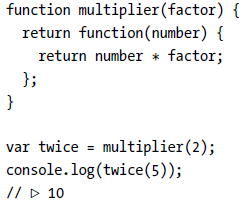
The upside is that this behavior can be used to have a function take “optional” arguments. For example, the following version of power can be called either with two arguments or with a single argument.



**Closure**



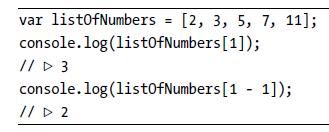
This feature—being able to reference a specific instance of local variables in an enclosing function—is called *closure*. A function that “closes over” some local variables is called *a* closure.



The explicit localVariable from the wrapValue example isn’t needed since a parameter is itself a local variable.

In the example, multiplier returns a frozen chunk of code that gets stored in the twice variable. The last line then calls the value in this variable, causing the frozen code (return number \* factor;) to be activated. It still has access to the factor variable from the multiplier call that created it, and in addition it gets access to the argument passed when unfreezing it, 5, through its number parameter.

**4 - DATA STRUCTURES: OBJECTS AND ARRAYS**

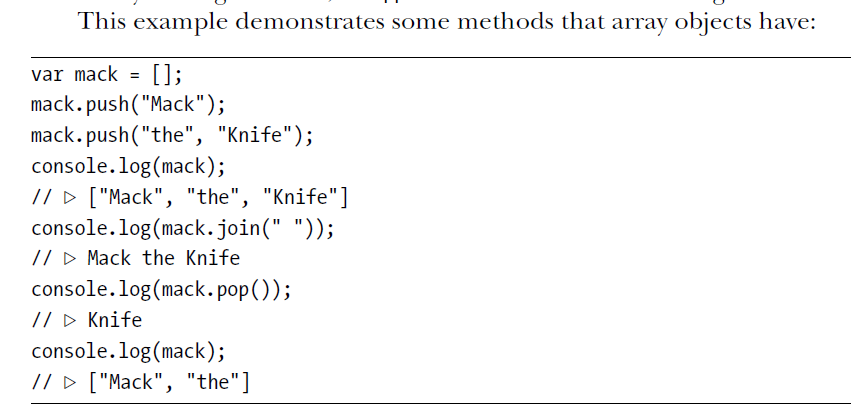


**Properties -** Almost all JavaScript values have properties. The exceptions are null and undefined. If you try to access a property on one of these nonvalues, you get an error.

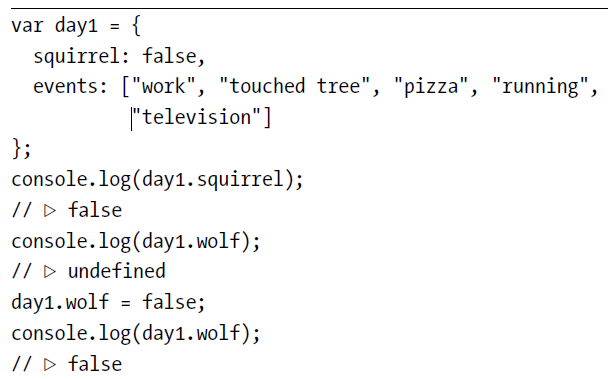
The two most common ways to access properties in JavaScript are with a dot and with square brackets. When using a dot, the part after the dot must be a valid variable name, and it directly names the property. When using square brackets, the expression between the brackets is *evaluated* to get the property name.

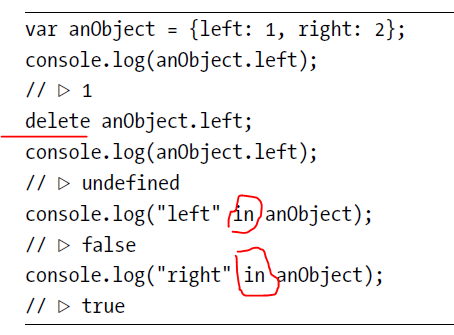
You typically write **array.length** because that is easier to write than **array["length"]**.

**Methods -** Properties that contain functions are generally called *methods* of the value they belong to. As in, “toUpperCase is a method of a string.”



**Objects -** Values of the type *object* are arbitrary collections of properties, and we can add or remove these properties as we please.

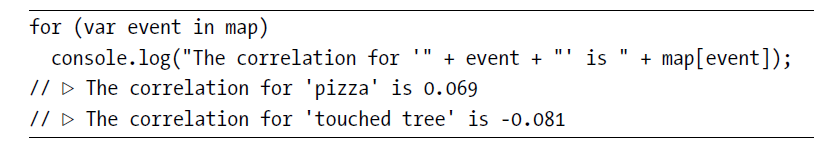




Arrays, then, are just a kind of object specialized for storing sequences of things. If you evaluate typeof [1, 2], this produces "object".

JavaScript provides a loop construct specifically for going over the properties of an object. It looks a little

like a normal for loop but distinguishes itself by the use of the word in.

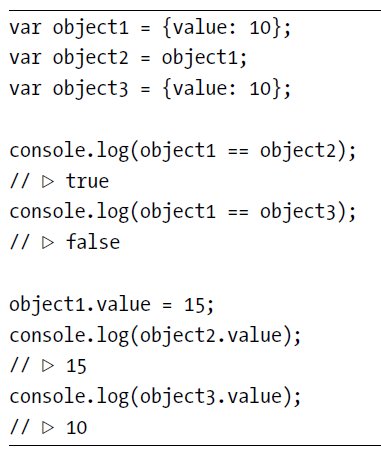


**Mutability -** numbers, strings, and Booleans, are all *immutable*—it is impossible to change an existing value of those types.

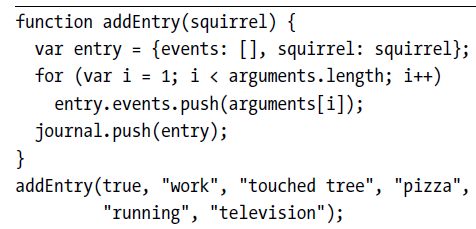
With objects, on the other hand, the content of a value *can* be modified by changing its properties.

When we have two numbers, 120 and 120, we can consider them precisely the same number, whether or not they refer to the same physical bits. But with objects, there is a difference between having two references to the

same object and having two different objects that contain the same properties. Consider the following code:

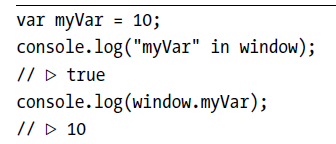


**The arguments Object -** Whenever a function is called, a special variable named arguments is added to the environment in which the function body runs. This variable refers to an object that holds all of the arguments passed to the function. The arguments object has a length property that tells us the number of arguments that were really passed to the function. It also has a property for each argument, named 0, 1, 2, and so on. If that sounds a lot like an array to you, you’re right, it *is* a lot like an array. But this object, unfortunately, does not have any array methods (like slice or indexOf), so it is a little harder to use than a real array.



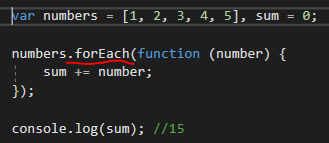
This version reads its first argument (squirrel) in the normal way and then goes over the rest of the arguments (the loop starts at index 1, skipping the first) to gather them into an array.

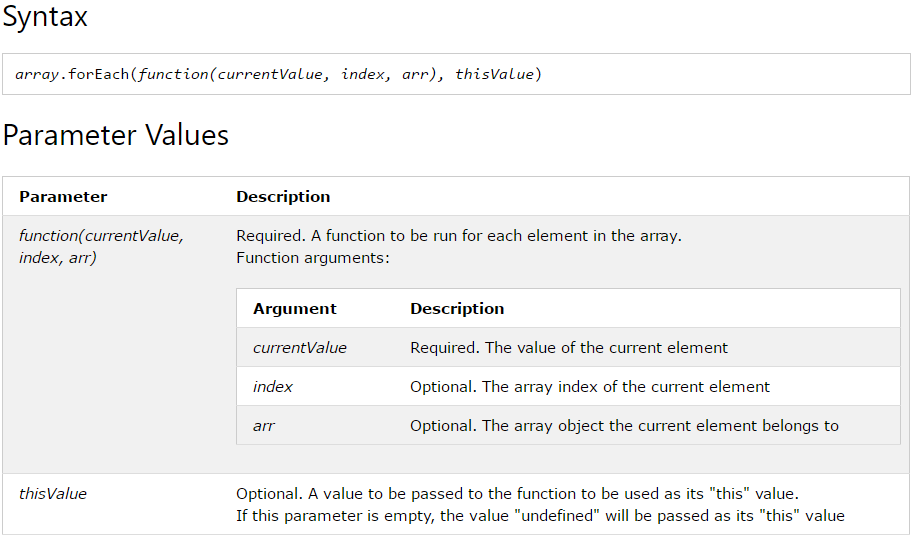
**The Global Object -** The global scope, the space in which global variables live, can also be approached as an object in JavaScript. Each global variable is present as a property of this object. In browsers, the global scope object is stored in the window variable.



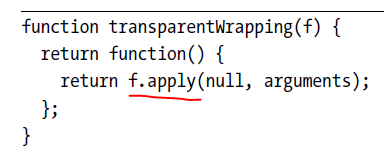
**5 - HIGHER-ORDER FUNCTIONS**

**Abstracting Array Traversal**





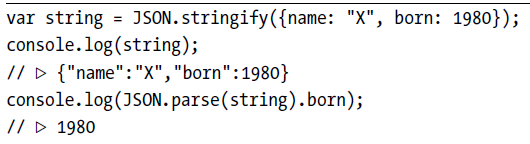
**Passing Along Arguments**



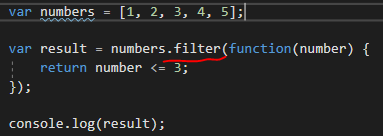
the function it returns passes all of the given arguments, and only those arguments, to f. It does this by passing its own arguments object to apply. The first argument to apply, for which we are passing null here, can be used to

simulate a method call. We will come back to that in the next chapter.

**JSON -** JSON.stringify and JSON.parse, that convert data from and to this format.

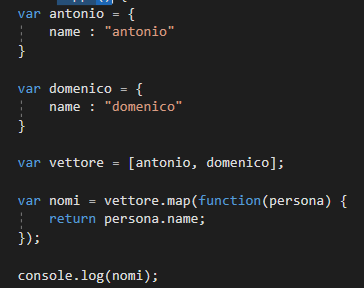


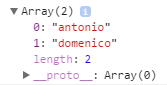
**Filtering an Array**



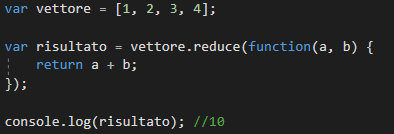


**Transforming with map -** The map method transforms an array by applying a function to all of its elements and building a new array from the returned values.





**Summarizing with reduce -**  You can reduce with any sort of operation that combines two values. Not just addition. And not just arithmetic operations.



Il primo valore di ‘a’ è 0, poi diventera il risultato della callback per le chiamate successive:

1 chiamata: a=0, b=1, risultato=1

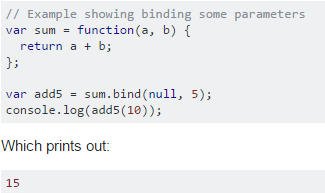
2 chiamata: a=1, b=2, risultato =3

3 chiamata: a=3, b=3, risultato = 6,

4 chiamata: a=6, b=4, risultato = 10

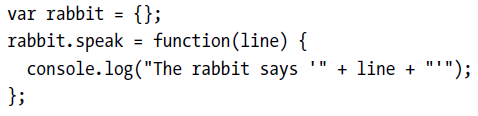
**Binding -** The bind method, which all functions have, creates a new function that will call the original function but with some of the arguments already fixed.

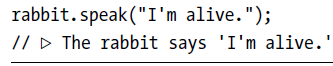
The first argument, where the example passes null, is used for method calls, similar to the first argument to apply. I’ll describe this in more detail in the next chapter.



**6 - THE SECRET LIFE OF OBJECTS**

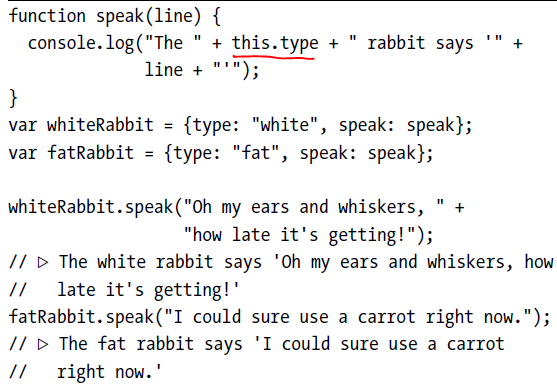
**Methods -** Methods are simply properties that hold function values.





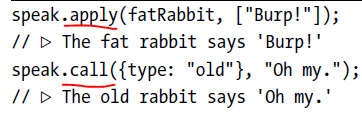
When a function is called as a method the special variable this in its body

will point to the object that it was called on.



Recall that the apply and bind methods both take a first argument that can be used to simulate method calls. This first argument is in fact used to give a value to this. There is a method similar to apply, called call. It also calls the function it is a method of but takes its arguments normally, rather than as an array.

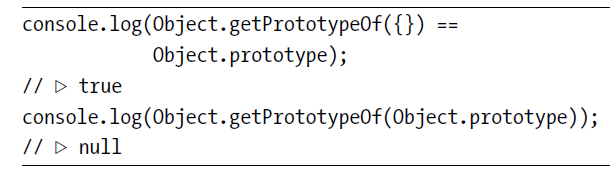
Like apply and bind, call can be passed a specific this value



**Prototypes -** In addition to their set of properties, almost all objects also have a *prototype*. A prototype is another object that is used as a fallback source of properties. When an object gets a request for a property

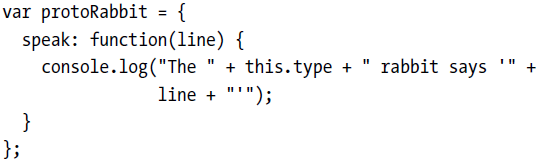
that it does not have, its prototype will be searched for the property, then the prototype’s prototype, and so on.

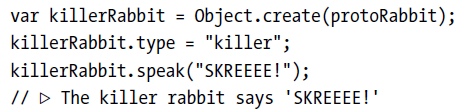
So who is the prototype of that empty object? It is the great ancestral prototype, the entity behind almost all objects, Object.prototype.



Many objects don’t directly have Object.prototype as their prototype, but instead have another object, which provides its own default properties. Functions derive from Function.prototype, and arrays derive from Array.prototype.

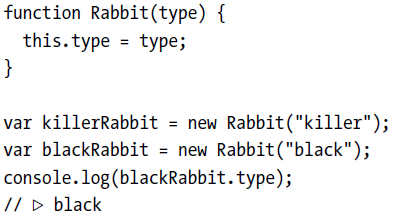
You can use Object.create to create an object with a specific prototype.

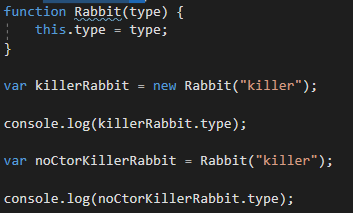


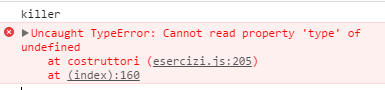


The “proto” rabbit acts as a container for the properties that are shared by all rabbits. An individual rabbit object, like the killer rabbit, contains properties that apply only to itself—in this case its type—and derives shared properties from its prototype.

**Constructors -** A more convenient way to create objects that derive from some shared prototype is to use a *constructor*. In JavaScript, calling a function with the new keyword in front of it causes it to be treated as a constructor. The constructor will have its this variable bound to a fresh object, and unless it explicitly returns another object value, this new object will be returned from the call. An object created with new is said to be an *instance* of its constructor. Here is a simple constructor for rabbits. It is a convention to capitalize the names of constructors so that they are easily distinguished from other functions.

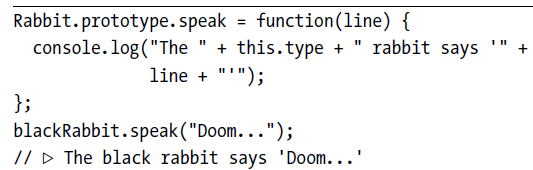




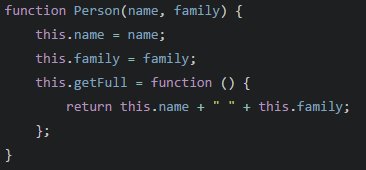


Constructors (in fact, all functions) automatically get a property named prototype, which by default holds a plain, empty object that derives from Object.prototype. Every instance created with this constructor will have this

object as its prototype. So to add a speak method to rabbits created with the Rabbit constructor, we can simply do this:



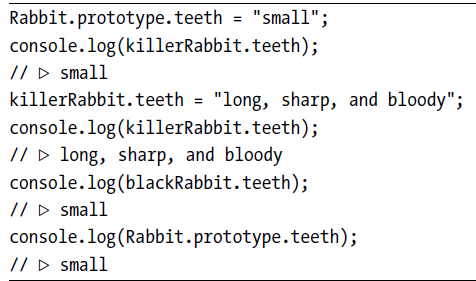
Da notare che un metodo potrebbe essere anche dichiarato nel costruttore invece che nel prototipo:



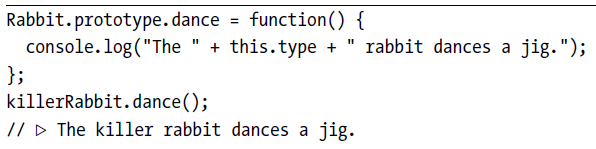
Yet the truth is, this approach might be wrong for many situations. In Javascript when you bind a method to the this keyword, you are providing that method to only that particular instance and it does not really have any relationship with an object instance of that constructor, pretty much like a static method. Keeping in mind that functions are first-class citizens in Javascript, we can deal with them just like objects, in this case we're only adding a property to an instance of a function object. Thats only part of the story, you must also know that any method attached via this **will get re-declared** for every new instance we create, which could affect the memory usage of the application negatively if we wish to create so many instances.

**Overriding Derived Properties -** When you add a property to an object, whether it is present in the prototype

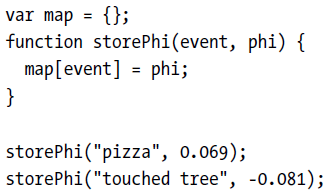
or not, the property is added to the object *itself*, which will henceforth have it as its own property. If there *is* a property by the same name in the prototype, this property will no longer affect the object. The prototype itself is not changed.

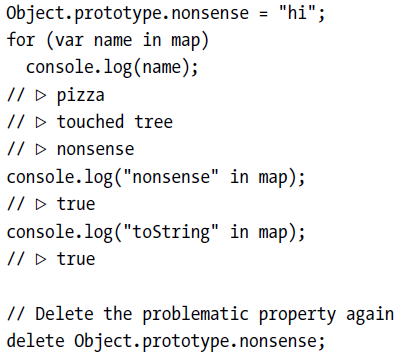


**Prototype Interference** A prototype can be used at any time to add new properties and methods to all objects based on it. For example, it might become necessary for our rabbits to dance.



That’s convenient. But there are situations where it causes problems

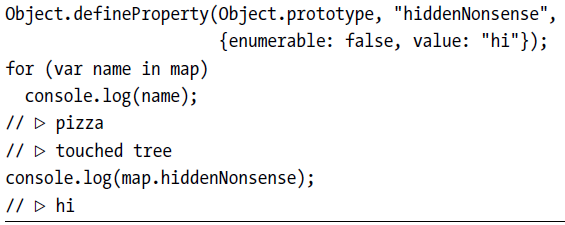




That’s all wrong. There is no event called “nonsense” in our data set. And there *definitely* is no event called “toString”.

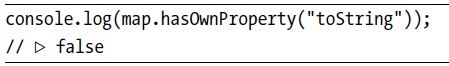
Oddly, toString did not show up in the for/in loop, but the in operator returned true for it. This is because JavaScript distinguishes between *enumerable* and *nonenumerable* properties.

All properties that we create by simply assigning to them are enumerable. The standard properties in Object.prototype are all nonenumerable, which is why they do not show up in such a for/in loop. It is possible to define our own nonenumerable properties by using the Object.defineProperty function, which allows us to control the type of property we are creating.

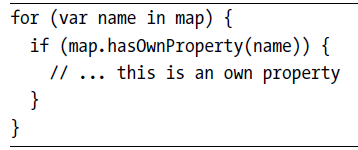


So now the property is there, but it won’t show up in a loop. That’s good.

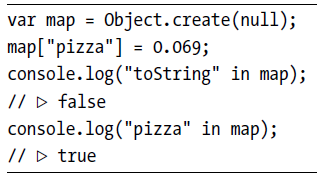
But we still have the problem with the regular in operator claiming that the Object.prototype properties exist in our object. For that, we can use the object’s hasOwnProperty method.



This method tells us whether the object *itself* has the property, withoutlooking at its prototypes. This is often a more useful piece of information than what the in operator gives us. When you are worried that someone (some other code you loaded into your program) might have messed with the base object prototype, I recommend you write your for/in loops like this:

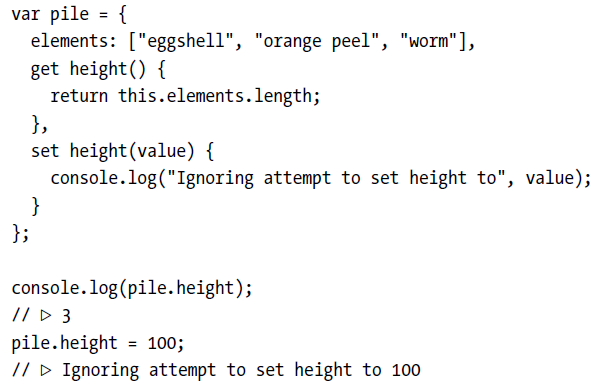


**Prototype-less Objects -**  What if someone registered the name hasOwnProperty in our map object and set it to the value 42? Now the call to map.hasOwnProperty will try to call the local property, which holds a number, not a function.

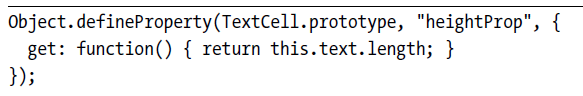


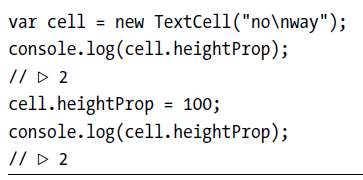
Much better! We no longer need the hasOwnProperty kludge because all the properties the object has are its own properties. Now we can safely use for/in loops, no matter what people have been doing to Object.prototype.

**Getters and Setters**



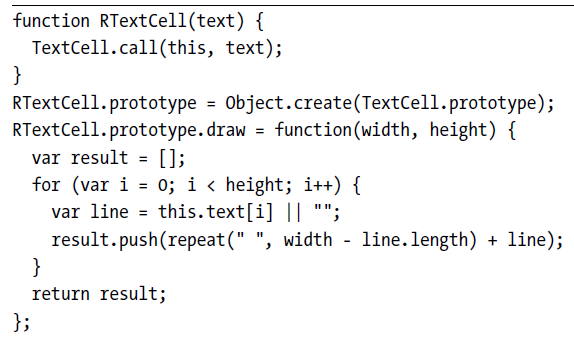
You can also add such a property to an existing object, for example a prototype, using the Object.defineProperty function (which we previously used to create nonenumerable properties).

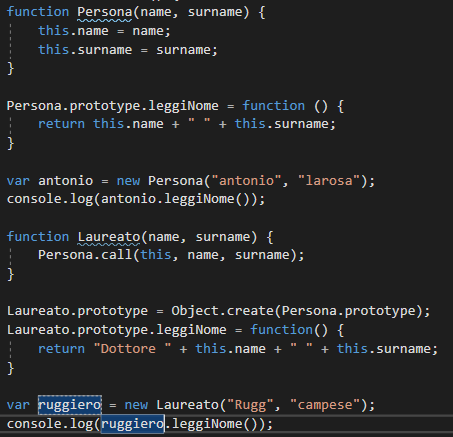




You can use a similar set property, in the object passed to defineProperty, to specify a setter method. When a getter but no setter is defined, writing to the property is simply ignored.

**Inheritance**





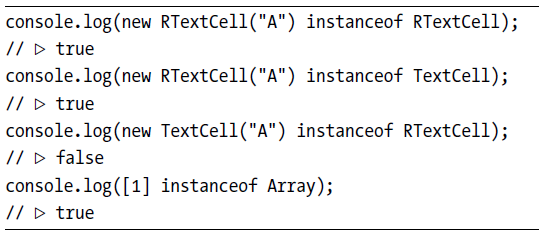


This pattern is called *inheritance*. It allows us to build slightly different data types from existing data types with relatively little work. Typically, the new constructor will call the old constructor (using the call method in order

to be able to give it the new object as its this value). Once this constructor has been called, we can assume that all the fields that the old object type is supposed to contain have been added. We arrange for the constructor’s

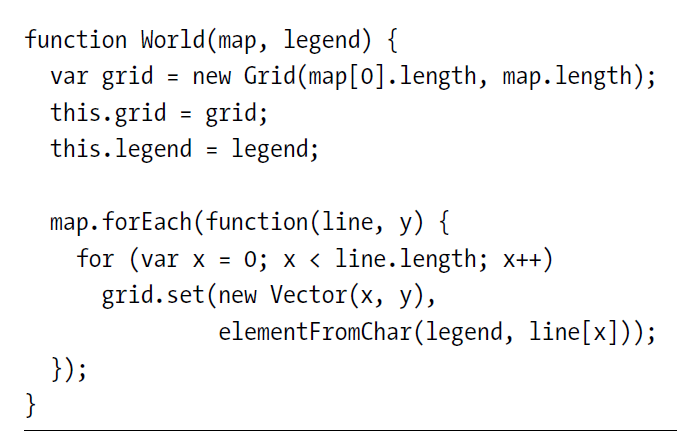
prototype to derive from the old prototype so that instances of this type will also have access to the properties in that prototype. Finally, we can override some of these properties by adding them to our new prototype.

**The instanceof Operator -** It is occasionally useful to know whether an object was derived from a specific constructor.



**7 - PROJECT: ELECTRONIC LIFE**

**this and Its Scope**



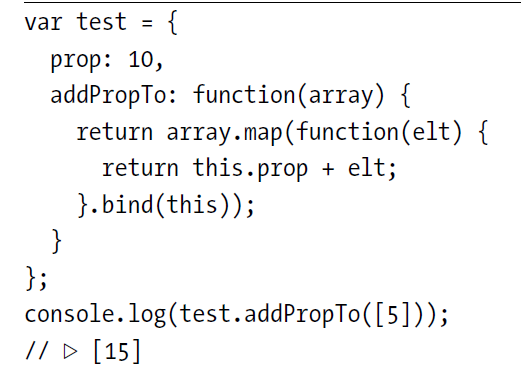
The World constructor contains a call to forEach. One interesting thing to note is that inside the function passed to forEach, we are no longer directly in the function scope of the constructor. Each function call gets its own this

binding, so the this in the inner function does *not* refer to the newly constructed object that the outer this refers to. In fact, when a function isn’t called as a method, this will refer to the global object.

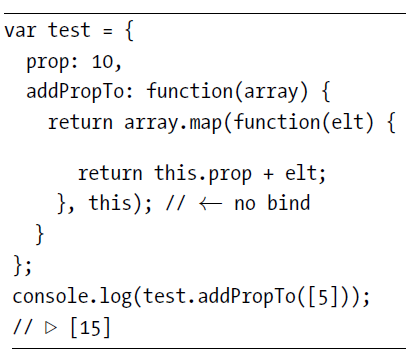
This means that we can’t write this.grid to access the grid from inside the loop. Instead, the outer function creates a normal local variable, grid, through which the inner function gets access to the grid.

A common pattern is to say var self = this and from then on refer to self, which is a normal variable and thus visible to inner functions.

Another solution is to use the bind method, which allows us to provide an explicit this object to bind to.

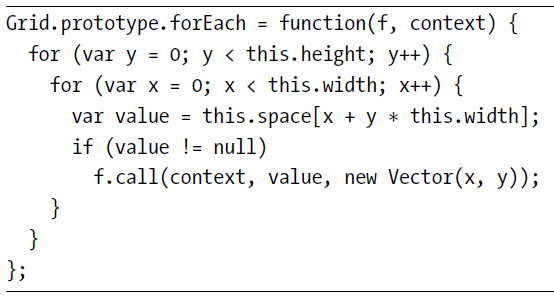


Most standard higher-order methods on arrays, such as forEach and map, take an optional second argument that can also be used to provide a this for the calls to the iteration function. So you could express the previous example in a slightly simpler way.

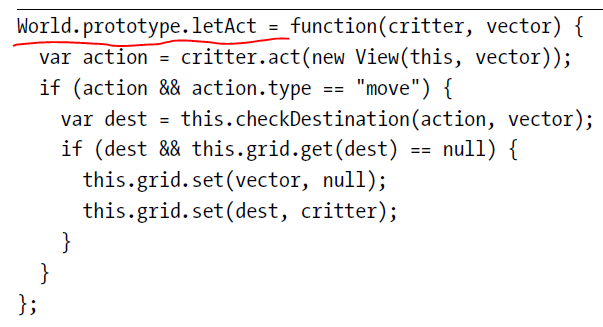


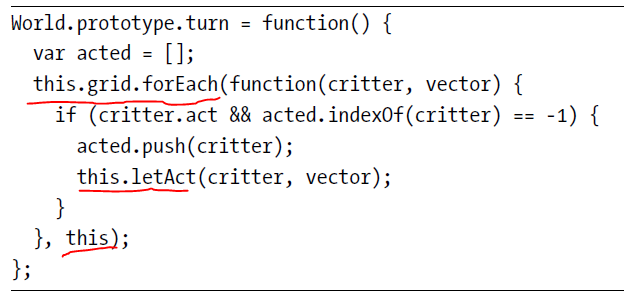
This works only for higher-order functions that support such a *context* parameter. When they don’t, you’ll need to use one of the other approaches.

In our own higher-order functions, we can support such a context parameter by using the call method to call the function given as an argument. For example, here is a forEach method for our Grid type, which calls a given function for each element in the grid that isn’t null or undefined:



Example:





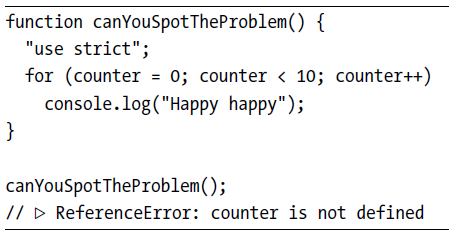
**8**

**BUGS AND ERROR HANDLING**

**Strict Mode**

JavaScript can be made a *little* more strict by enabling *strict mode*. This is

done by putting the string "use strict" at the top of a file or a function body.



Normally, when you forget to put var in front of your variable, as with

counter in the example, JavaScript quietly creates a global variable and uses

that. In strict mode, however, an error is reported instead. This is very helpful.

It should be noted, though, that this doesn’t work when the variable

in question already exists as a global variable, but only when assigning to it

would have created it.

Another change in strict mode is that the this binding holds the value

undefined in functions that are not called as methods.

When making such a

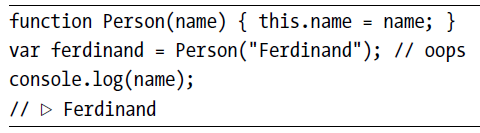
call outside of strict mode, this refers to the global scope object. So if you accidentally call a method or constructor incorrectly in strict mode, JavaScript will produce an error as soon as it tries to read something from this, rather than happily working with the global object, creating and reading global variables.

variables.

For example, consider the following code, which calls a constructor

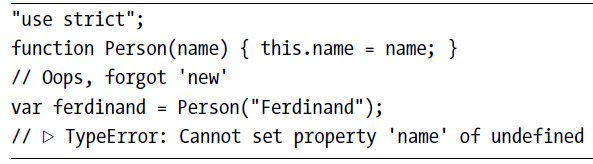
without the new keyword so that its this will *not* refer to a newly constructed

object:

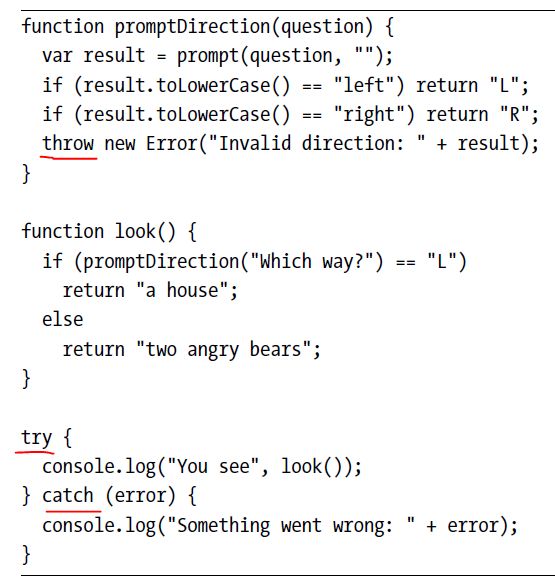


So the bogus call to Person succeeded but returned an undefined value

and created the global variable name. In strict mode, the result is different.



**Exceptions**



In this case, we used the Error constructor to create our exception value.

This is a standard JavaScript constructor that creates an object with a message

property. In modern JavaScript environments, instances of this constructor

also gather information about the call stack that existed when the exception

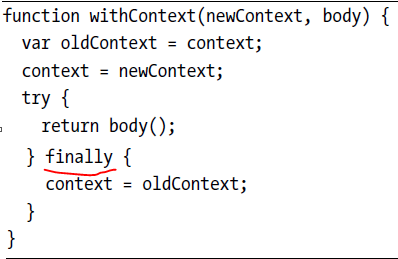
was created, a so-called *stack trace*. This information is stored in the

stack property and can be helpful when trying to debug a problem: it tells us

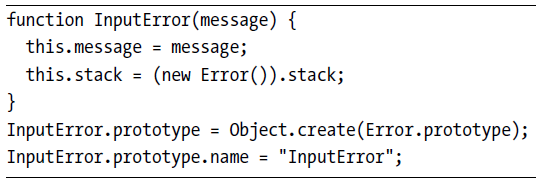
the precise function where the problem occurred and which other functions

led up to the call that failed.

**Cleaning Up After Exceptions**



**Selective Catching**



The prototype is made to derive from Error.prototype so that instanceof

Error will also return true for InputError objects. It’s also given a name property

since the standard error types (Error, SyntaxError, ReferenceError, and so on)

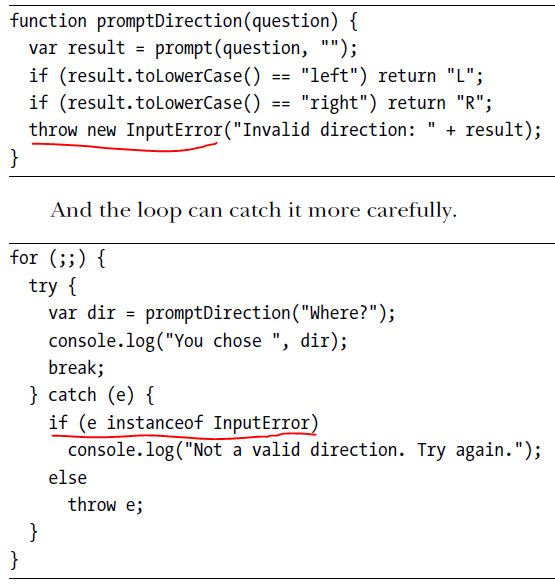
also have such a property.

The assignment to the stack property tries to give this object a somewhat

useful stack trace, on platforms that support it, by creating a regular error

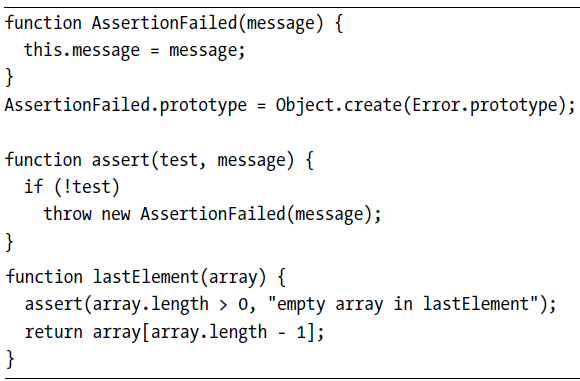
object and then using that object’s stack property as its own.

Now promptDirection can throw such an error.



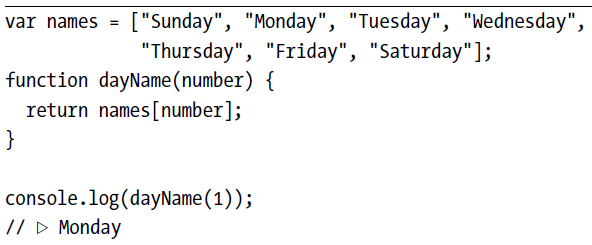
**Assertions**

*Assertions* are a tool to do basic sanity checking for programmer errors.



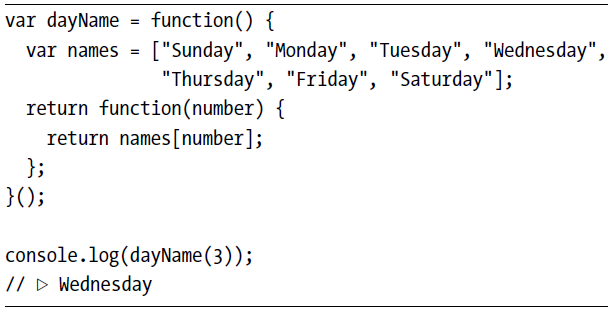
**10 - MODULES**

**Using Functions as Namespaces**

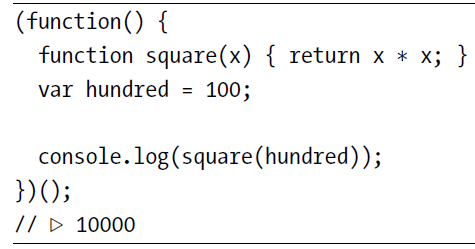


The dayName function is part of the module’s interface, but the names variable

is not. We would prefer *not* to spill it into the global scope. We can do this:



We can use a similar pattern to isolate code from the outside world entirely.



This code simply outputs the square of 100, but in the real world it could

be a module that adds a method to some prototype or sets up a widget on a

web page. It is wrapped in a function to prevent the variables it uses internally

from polluting the global scope.

Why did we wrap the namespace function in a pair of parentheses? This

has to do with a quirk in JavaScript’s syntax. If an *expression* starts with the

keyword function, it is a function expression. However, if a *statement* starts

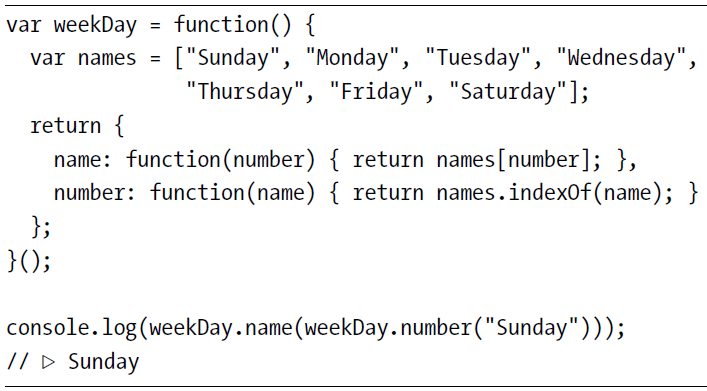
with function, it is a function *declaration*, which requires a name and, not being

an expression, cannot be called by writing parentheses after it. You can

think of the extra wrapping parentheses as a trick to force the function to be

interpreted as an expression.

**Objects as Interfaces**



For bigger modules, gathering all the *exported* values into an object at

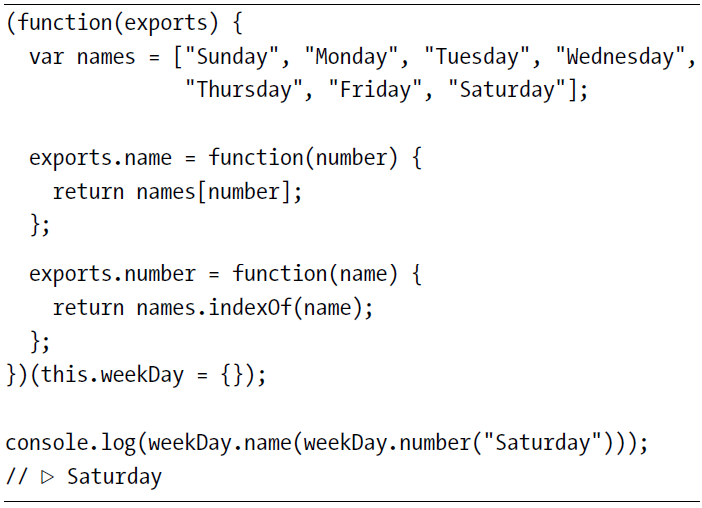
the end of the function becomes awkward since many of the exported functions

are likely to be big and you’d prefer to write them somewhere else,

near related internal code. A convenient alternative is to declare an object

(conventionally named exports) and add properties to that whenever we are

defining something that needs to be exported.



**Detaching from the Global Scope**

The previous pattern is commonly used by JavaScript modules intended for

the browser. The module will claim a single global variable and wrap its code

in a function in order to have its own private namespace. But this pattern

still causes problems if multiple modules happen to claim the same name or

if you want to load two versions of a module alongside each other.

With a little plumbing, we can create a system that allows one module

to directly ask for the interface object of another module, without going

through the global scope. Our goal is a require function that, when given a

module name, will load that module’s file (from disk or the Web, depending

on the platform we are running on) and return the appropriate interface

value.

This approach solves the problems mentioned previously and has the

added benefit of making your program’s dependencies explicit, making it

harder to accidentally make use of some module without stating that you

need it.

**Evaluating Data as Code**

Function constructor. This takes two arguments: a string containing a comma-separated list of

argument names and a string containing the function’s body.

