Adventures in JavaScriptLand

By

Thomas Walker Lynch
2019-04-20

Thomas.Lynch@ReasoningTechnology.com

edition 0.1

Introduction

It would not be proper, for some reasons, to trouble the reader with the particulars of our adventures in the intervening seas; let it suffice to inform him, that in our passage from thence to the East Indies, we were driven by a violent storm among upper management. By an observation, we found ourselves immersed in thousands of lines pulled from Github. Twelve of our crew were dead by immoderate labour and ill management; the rest were in a very weak condition. On the 5th of November, which was the beginning of summer in these parts, the weather being very hazy, we were ordered to adopt the use of JavaScript. We were driven directly upon it, and immediately the team split. Only six of the crew, of whom I was one, remained. We therefore trusted Emacs JavaScript mode browse-url-of-buffer, and set down to study that which we were betrothed. I cannot tell; but conclude the others were all lost. For my own part, I swam as fortune directed me, and was pushed forward by only the will to know. When I was almost gone, and able to struggle no longer, I found myself within my depth; and by this time the storm was much abated. The declivity was so small, that I Googled Stack Overflow merely for a month before finding my footing. I then advanced forward one small step at a time. This is an object, then this is a prototype, etc., but I could not discover any sign of intelligible output in the browser. After the Nth allniter I was in so weak a condition, that I did not observe Firefox any further. I was extremely tired, and with that, and about half a gallon of coffee that I drank but did not help, as I left the moorings of my desk, I found myself much inclined to sleep. I lay down on the grass outside the lobby, which was very short and soft, where I slept sounder than ever I remembered to have done in my life, and, as I reckoned, about nine hours; for when I awaked, it was mid afternoon. I attempted to rise, but was not able to stir: for, as I happened to lie on my back, I found my arms and legs were strongly fastened on each side to the ground; and my hair, which was long and thick, tied down in the same manner because my colleagues had played a practical joke. And this is how I arrived in JavaScriptLand.

First Impressions

JavaScriptLand is populated by people who have strange customs and culture. Their language appears to be of Proto Indo-CS with common roots to our own language at some point in antiquity. Consequently there are many false cognates. Take the name of the language itself. *JavaScript,* also known as *JavaScriptLandese*, is not a script form of what we call *Java*, nor is it the language spoken by the Javanese. Far far away from it. These languages are completely unrelated. The common prefix ‘java’ is merely an accident of history.

As additional examples of false cognates, what the JavaScriptLand people call *objects*, are not the *objects* of ‘object oriented programming’, rather they are merely association lists. What they call a *prototype* is not a type declaration header for a function. Nor is it a model version of something being built. Rather, it is more closely related to what we Homelander’s would call an abstract type definition, i.e. a list of functions that define a type. Given all this, it is probably not surprising to learn that even what the JavaScriptLand people call a *function*, is not what you or I here in Homeland would call a function. Rather it is a *functor*, i.e. a singleton class masquerading as a function.

It is important for you to know that the people of JavaScriptLand very much want to be our friends, so they have been adopting into their language syntax constructs we use in our own language; however, due to the cultural and perceptual differences, it hasn't been a perfect process.

This document is not intended to be a guide for learning JavaScript. If you use it for that purpose (surely along with other material) and it works out, great, I would like to hear about it. Rather my intention is to give a commentary on my adventure in JavaScriptLand. If you have read documents written in JavaScriptLandese while thinking the false cognates had the same meaning as the corresponding words in Homelandese, then you no doubt became confused. If so then this adventure story was written for you. In these pages I will share my impressions and attempts at trying to understand this strange language of JavaScriptLandese. In the process I hope to shed some light on their foreign tongue and to dispel some common myths.

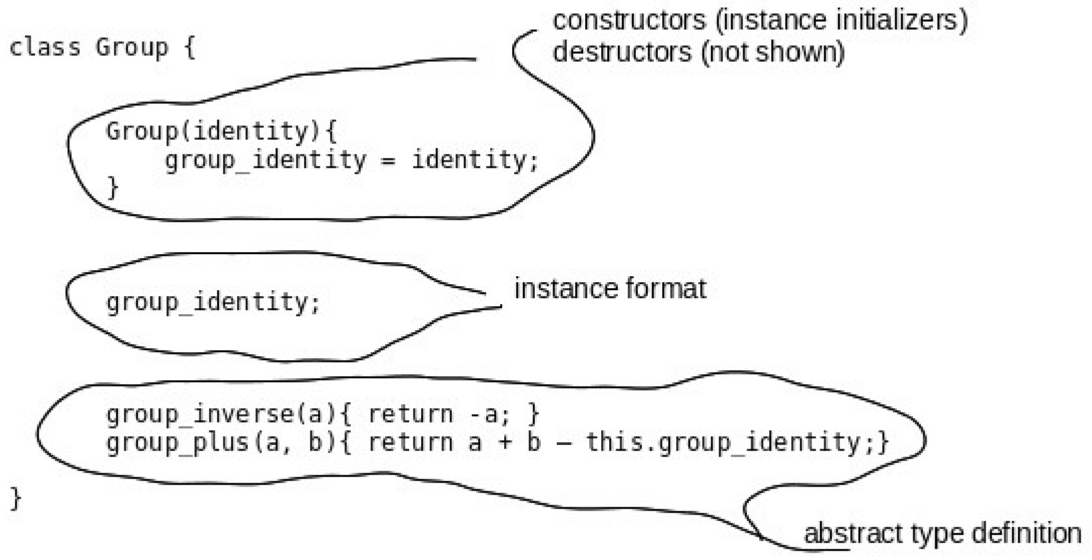
Perspectives on Homeland

One of the wonderful side effects of learning a foreign language is that it lends perspective into one's own language. Hence, this is where I begin, not in a foreign land, but with a review of our own object oriented programming. In the silly example just below I show some pseudo code for a class definition not too much unlike how it would be written in Java, C++, Python or a number of other languages.

```
class Group {

       Group(identity){
          group_identity = identity;
       }

       group_identity;

       group_inverse(a){ return -a; }
       group_plus(a, b){ return a + b – this.group_identity; }

}
```

Figure 1: Psuedocode for a Conventional Class Declaration

We Homelanders will sometimes use the terms 'class definition' interchangeably with that of 'type definition'. We say that the type of an instance is the same as the name of the class. For example, an instance of 'class Animal' is said to be an 'Animal' type, and for our running example, an instance of 'class Group' has the 'Group' type. A class definition is a concrete type definition because it describes the format for instances, how to initialize them, and provides an abstract definition of type in the form of a function list for operating on the instances. As such, our class definition for *Group* has three distinct parts,



```
class Group {

    Group(identity){
        group_identity = identity;
    }

    group_identity;

    group_inverse(a){ return -a; }
    group_plus(a, b){ return a + b – this.group_identity;}
}
```

After we define a class we will typically make multiple instances of the class using an operator such as *new*. An instance takes up memory typically on the stack or on the heap. The format of the instance is given by the non-static portion of the class definition. In this example there is only one non-static member, that being *group_identity*. A C or C++ programmer might imagine an instance format as being described by a *struct.* In this example such a *struct* would have one field called *group_identity*.

The job of a constructor is to initialize an instance after it is allocated. Later when the instance is to be deallocated there might be some cleanup work to be done. This is more common in languages that don't have garbage collection, but it can happen in either case. The constructor here is *Group(),* we recognize it as such because it has the same name as the class. This is a common convention, though not one followed by JavasScriptLanders. JavaScriptLanders like to call a constructor *constructor*, which seems logical enough. However, they do not have function overloading, so this convention causes them some trouble when there are multiple constructors. If a language happens to support it, a destructor is called to perform the cleanup just before an instance is deallocated. Our example does not define a destructor, and that is OK for our purposes here because JavaScript doesn't have them.

The remaining part of the class definition consists of the non-constructor, non-destructor, static members. In this example there are only two, *group_inverse()*, and *group_plus()*. A list of functions which tells you everything you can do with instances of a given format is known as an abstract type definition. Say for example, if someone asked you what an integer is, you could reply by giving a list of all the functions that operate on integers and say, "it is that thing that these functions will operate on". In our example there are only two things you can do with a type group instance, that is *group_inverse()*, and *group_plus()*.

We Homelanders are accustom to class declarations automatically implying a namespace during inheritance. That is to say we might want to refer to *group_identity* as *Group::group_identity*, or *group_inverse()* as *Group::group_inverse()*. JavaScriptLanders do not have such a custom. This might lead to some problems for Homelanders, which is why I manually created the namespace in advance by prefixing identifiers with the name of the class followed by an underscore. No doubt JavaScriptLanders will see such a notation as foreign to them, so I discuss this issue and its implications at more length in a later chapter dedicated to the subject.

During compilation or interpretation the class definition and instances will be held in data structures, and these will be accounted for by using a symbol table. Such a table might list each symbol name given to an instance in a first column, then the type of the instance in a second column, and then surely a reference to the named instance in a third column. For compiled languages with only static type, the information in the symbol table will be 'compiled away' by moving the references into the generated code. For interpreted languages, the symbol table might be built into the interpreter and modified and used at run time.

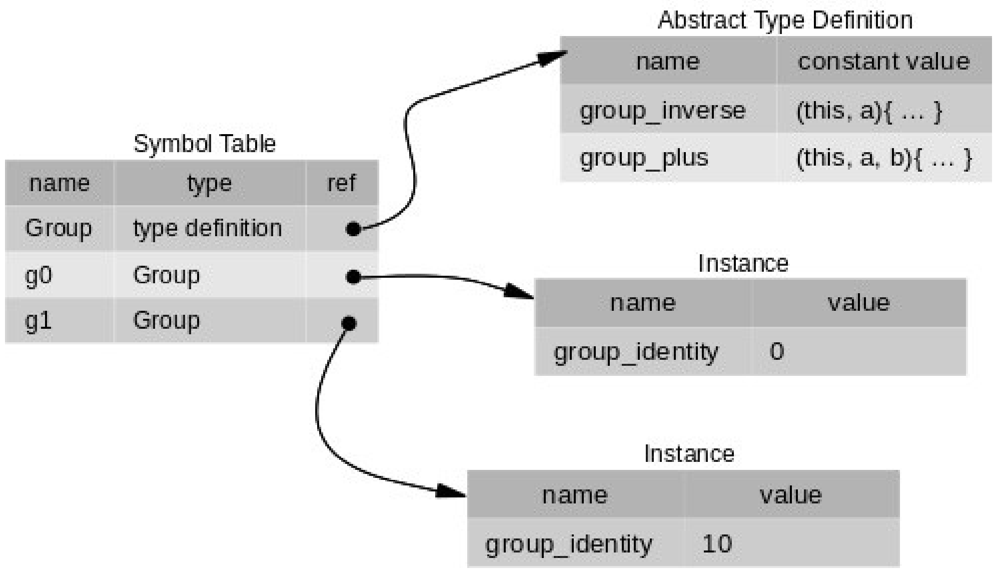


Figure 2: Conventional View of Instances and Abstract Type Definition

Shown here are two instances of type *Group*, an abstract type definition for *Group*, and a symbol table.

The name fields which are shown in the first column of the *Abstract Type Definition* and in each *instance,* would also typically be in the symbol table. Accordingly, each field name, *group_inverse*, *group_plus* and *group_idenity,* would correspond to an address offset for finding the respective fields in the respective instances. Perhaps it would look something like this:

name	type	ref
Group	type definition	[]
g0	Group	[]
g1	Group	[]
group_inverse	offset	0
group_plus	offset	8
group_identity	offset	0

Figure 3: Symbol Table Including the Field Offsets

However I didn't want to muddy the diagram with such details, so I just wrote the field names next to their values in the referenced instances. By coincidence, this also happens to be the preferred convention in JavaScriptLand. This clarification in the diagram did lead to some redundancy for the *Group* instances because I had to write 'group_identity' twice, once in each instance, instead of just having it listed just once in the symbol table. It is of little consequence here, but in a situation where there are many, perhaps thousands, of instances this would add up.

When it comes time to parse the code and to call a class method, we tack the name of the method onto the end of the name of a variable holding an instance. For example, by using a dot notation. Because the method functions live in the shared class definition, the methods themselves do not have any built in means for locating any particular instance. Methods can have built in references to static data, but not to instance data. So, when a method is called 'on an instance', a reference to the instance is passed in as one of the arguments. Sometimes this is not shown to the programmer, but it will be there hidden under the hood. Historically such an instance reference has been called *this*.

```
g1.group_plus(a, b);
```

Figure 4: Calling a Class Method on a Class Instance

Here *g1* is the instance name. We will use the symbol table to look up the instance name to find a reference to the instance. In this same table we will also find the instance's type name, in this case *Group*. We will then look up the type name and find the abstract type definition. Within the abstract type definition we will look up the function name, *group_plus*, and find the code for it. If we are compiling this code, we will then generate the code for calling *group_plus()*. Otherwise, if we are interpreting it, we will then call *group_plus()*. In either case, the first argument given to *group_plus()* will be a reference to the instance we looked up. In this example that will be a reference to *g1*. (If we had been working on another instance we would have supplied that one instead.) We will then look up the remaining arguments in the symbol table and provide those also.

While we Homelanders are gathered here and having such a nice discussion about our own esoteric customs it will be convenient to also describe our custom of ‘inheritance’ , especially for the sake of the younger among us who, due to rebellious attitude that is so common among youth, did not take their studies of our customs so seriously, but now wish they had.

Herein I will say that the class we inherit from is the *parent*, and the class that we inherit into is the *child*. In the many dialects found in our Homeland people will refer to these as *base class* and a *derived class*, as *class* and an *extension*, or even as *class* and a *specialization*. I do not mean to belittle these other dialects, rather I just desire to pick one set of terms for sake of clarity or presentation. Now consider we have the class *Ring* which inherits from *Group*, again using psuedo-code:

```
class Ring : Group {
       Ring(ring_identity){
           ring_identity = identity;
       }
       ring_identity;
       ring_inverse(a){
           return ring_unity/(a – group_identity) + group_identity;
       }
       ring_times(a,b){
           return
           (a - group_identity) * (b – group_identity) + group_identity;
       }
}
```

Figure 5: A Child Inheriting from the Parent Group

We have a few options for implementing inheritance. One option would be to copy the parent definition and then append the new child parts, so as to make an entirely new child class definition. Then child instances would be instantiated from this child class definition as is done for any other class definition. Though there is a problem with this approach when we ask the question: “who is the parent of this child?” Perhaps we might build separate inheritance documentation and file that off with a mediating agency, who could then be queried to provide answers to such questions.

A second option would be to create a child class definition that merely refers to the parent class definition. With this approach we would traverse the definition graph when allocating and initializing an instance. Usually such a traversal will be simple. With this approach all the references required for answering inheritance questions will then be available in the type definition graph without any additional structure. Here is an expanded diagram showing the Ring inheritance:

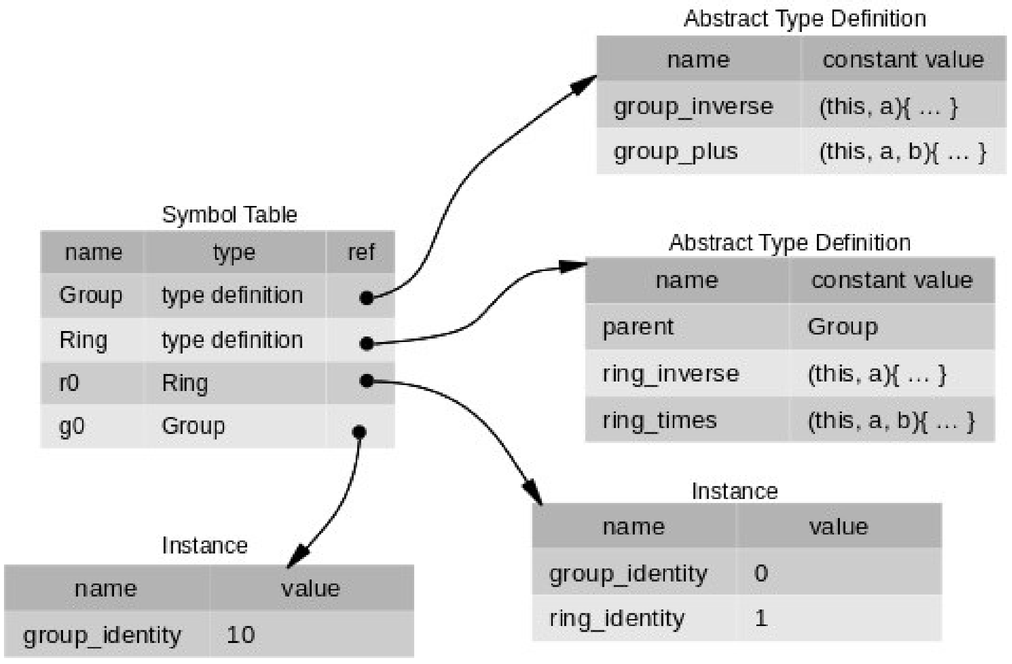


Figure 6: Conventional View of Instances and Class Definition with Inheritance

Here the abstract type definition for *Ring* includes a field for a reference to a parent. If instead we allowed this to be a list of references we could have multiple inheritance. The code that allocates a *Ring* instance will follow the inheritance hierarchy and make an allocation that is large enough to hold both the *Group* and *Ring* non-static members. Then the *Group* constructor and methods will be called with a *this* pointer to the group part so that the original group code will still work, where as the *Ring* constructor and methods will receive a *this* pointer to the whole of the instance. It all works because when Ring is compiled we know its entire type definition including that of its parent. Without multiple inheritance both *Group* and *Ring* start at the same address, so there will only be one *this* pointer.

In this example the parent field of the *Ring* type definition is symbolic, and we go back to the symbol table to get the reference to the *Group* type definition. As an optimization perhaps we could just put the reference to the parent directly into the *parent* field of the Ring type definition. And since we are taking shortcuts, we might as well also add fields in the instances to point back to their abstract types. Here is a modified diagram with the shortcuts:

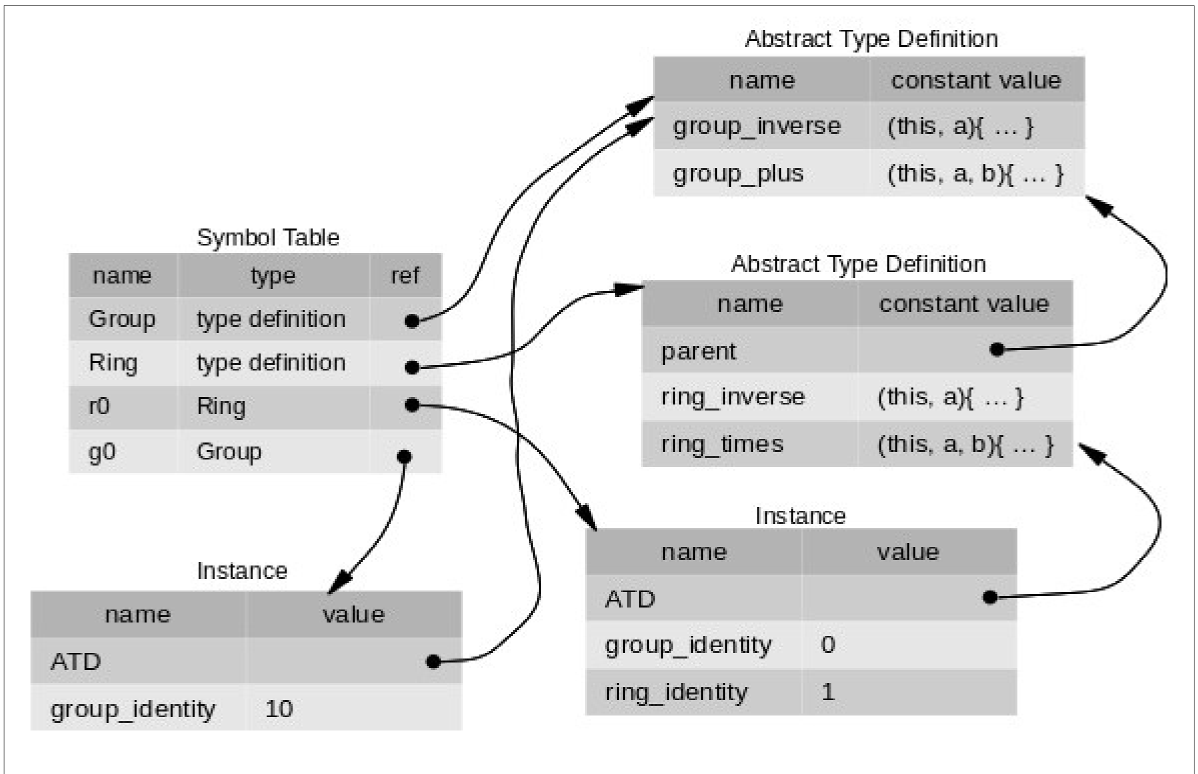


Figure 7: Conventional View with Shortcut References

Well no doubt you are already familiar with these concepts due to having grown up in Homeland, but it doesn't hurt any to review them so they are fresh. As it turns out, perhaps due to the laws of nature, these same concepts apply in JavaScriptLand. In fact, if we take Figure 7 and and rename the *ATD* fields to __*proto*__ and instead of saying "*abstract type*" we say "*prototype*" – we will be speaking JavaLandese. Though it is true that the words '__*proto*__' and '*prototype*' are strange and different, of course we can't expect people in a foreign land to use the same words for the same things as we do.

What is it I hear that person in the back row asking? .. "How do you pronounce, __*proto*__?" Oh yes, the foreign accents and special characters can throw a person off. In the international phonetic alphabet this is pronounced as "dʌndər proʊtoʊ". Don't worry about getting all this at first, we will go over it in more detail in later sections.

From here forward, we will endeavor to learn JavaScriptese through the total immersion method. No more pseudo code!

Read-Eval-Print

To open the JavaScript console in Chrome or Firefox, hit F12 on the keyboard. You will find yourself in a small integrated development environment, with the console prompt shown in the window of one of the panes, probably the one on the bottom. The JavaScript console is much like that for other interpreted languages. There is a reader that reads what was typed, evaluates it, and then prints a result. Of course the nifty part is that the evaluation of JavaScript language statements can have the side effect of rendering HTML, SVG, or canvas work in the browser, which is typically the whole point of speaking the language. In this sense it reminds me of the postscript language and its interpreter. Surely these cultures are somehow related.

Primitive Types

The JavaScript language primitive types are straightforward enough:

```
undefined, null, boolean, number, or string
```

Figure 8: JavaScript Primitive Types

undefined and *null* are singleton types, i.e. they are their own instances. The literals *undefined* and *null* are recognized by the reader. An instance of *boolean* type will have a value of either *true* or *false*. These are also literals recognized by the reader. During the evaluation of Boolean expressions, *undefined* and *null* play the same role as *false,* while an instance of anything else may be used to play the role of *true*.

Either single or double quotes may be used for strings. There is no character type. So for example, ‘c’ is an instance of string type. In some situations numbers are considered to be strings, as discussed in the section on the Array type further on. Sometimes strings do not require being quoted, as discussed in the next section on association lists.

JavaScript performs short circuit evaluation of Boolean expressions. When evaluating a Boolean expression it will return the instance that tripped the short circuit.

```
let x = 5;
true && x;  // returns 5
x && false; // returns false
true && x && null && true; // returns null
true && x && undefined;  // returns undefined
false || x || true; // returns 5
true || x || false;  // returns true
```

Figure 9: Boolean Expression Evaluation

Association Lists

The JavaScript reader appears to turn all input into either a primitive type instance or into an association list instance. To give an association list directly to the reader enter it using the following syntax:

```
{ <property_name> : <property> , … }
```

Figure 10: Reader Syntax for an Association List

The braces are literal punctuation of course. The list is comma separated. A *property_name* is separated from the corresponding value with a colon. Blank space is ignored. The printer on my browser puts the colon immediately after the *property_name*, then puts a space after the colon, and then prints the *property*. This makes it look like the colon is part of the *property_name*, but it is not.

Properties are read back from the association list using either a dot notation or a bracket notation. Within the association list a *property_name* may be either unquoted as per the dot notation, or quoted as per the bracket notation.

```
> x = {a:1, 'b c':2}
{a: 1, b c: 2}

> x.a
1

> x.'b c'
VM124:1 Uncaught SyntaxError: Unexpected string

> x['b c']
2
```

Figure 11: Dot and Bracket Access Operators

There are conventional *property_names* used by the language which are probably not a good idea to use on your objects unless you understand the implications. These include '__proto__', 'prototype', and 'constructor'.

The reader requires that any *property_name* to be used with the dot notation accessor be limited to letters, digits, and the special characters '_' and '$'. Such a *property_name* can be a list of only digits, but if it is not just digits, it can not start with a digit. Letters are limited to certain ranges in Unicode. These dot accessor property_names are taken literally, i.e. are not evaluated. I've been typing at my Chrome browser while trying to mess up the dot nation by using keywords, I've not hit a bad case yet. Like all foreign languages JavaScriptLandese has many seemingly arbitrary grammar rules that just have to be memorized.

```
> z = {if:0, false:1}
{if: 0, false: 1}
> z.if
0
> z.false
1
```

Figure 12: Association List Dot Access

The rules for *property_names* used with the bracket notation are much simpler. With the bracket accessor a *property_name* is either a quoted string or a string of digits optionally without quotes.

A *property_value* may be anything which we can make an instance of, which comes down to being a primitive type or an association list.

The JavaScriptLanders call association lists *objects*. This term can be confusing, especially as JavaScript now includes *class* keywords and other trappings of object oriented programming. In this document I will continue to use the term ‘association list’ for association lists.

The astute reader might think to ask: “Then what do you get when you *new* a *class*?” Well you get an *instance* of course. This latter nomenclature is actually something I already use when talking about object oriented programming, because the definition of what is an *object* has become so vague in our Homeland.

Although the documentation for JavaScript talks about *property_name* and *property*, the standard functions are named using the terms *key* and *value*. In yet other places I’ve seen the term *property* used to refer to the property name while the term *value* was used to refer to the *property*. Note that JavaScript has added a distinct *Map* type for larger data collections, and it is said to map *keys* to *values*.

Here is another example:

```
let io = {stdin:0, stdout:1, stderr:2};
let x = io.stdin;
let y = io[‘stdout’];
let z = io.stderr
```

Figure 13: Association List Access Example

In this example, the reader will recognize and build the association list declared between the braces, and then place a reference to it in the variable *io*. At the end of this code snippet *x* will be 0, *y* will be 1, and *z* will be 2.

An association list may be accessed using either the dot notation or the bracket notation. If an unknown *property_name* is given, then the accessor returns *undefined*.

The keyword *let* declares a new variable to the interpreter. Variables declared with *let* only exist while they are in block scope. Block scope occurs between braces.

```
> { let x = 1; console.log(x); {let x = 3; console.log(x); } console.log(x)}
console.log(x)
1
3
1
VM326:1 Uncaught ReferenceError: x is not defined
```

Figure 14: let Scoping

Old timer JavaScriptLanders used the *var* keyword for declaring variables. It still works. Variables declared as such will be available anywhere within function scope presumably after they are declared.

JavaScript provides an association list whose properties are functions for operating on association lists. It is called *Object*. One will find in this association list things such as

Object.keys(*alist*), which will return a list of all the property names used in *alist*, and *Object.values(alist)* which returns all the properties:

```
> console.log(Object.getOwnPropertyNames(Object));
["length", "name", "prototype", "assign", "getOwnPropertyDescriptor",
"getOwnPropertyDescriptors", "getOwnPropertyNames", "getOwnPropertySymbols",
"is", "preventExtensions", "seal", "create", "defineProperties",
"defineProperty", "freeze", "getPrototypeOf", "setPrototypeOf",
"isExtensible", "isFrozen", "isSealed", "keys", "entries", "values",
"fromEntries"]
```

Figure 15: Object Methods (and a couple of other properties)

Here I've used the *Object* method *getOwnPropertyNames* to list the property names used in the association list called *Object*. Yes, that is self referential, but it works just fine. As we will discuss at length in the next section and later sections, the property, *prototype*, is a special purpose thing that is not a method, and as we will see this is part of creating an inheritance hierarchy. The word 'Own' in this function name means we don't want to traverse an inheritance hierarchy, but just want the names of the properties found directly on the referenced association list.

```
> let io = {stdin:0, stdout:1, stderr:2};
> Object.keys(io)
["stdin", "stdout", "stderr"]
> Object.name(io)  // name is a value, not a function
Uncaught TypeError: Object.name is not a function
> io.name  // io has no such property as name
undefined
> Object.name
"Object"
```

Figure 16: Various Object Calls

JavaScript association lists may also have properties that are used by the interpreter and can not be manipulated by the programmer. These are said to be *internal* properties. In the Chrome console these are show inside of double brackets [[…]]. JavaScript association lists may also be given access permissions, note the *isExtensible()*, *isFrozen()*, and *isSealed()* methods. Furthermore getters, setters, read-only status, and yet other properties may be assigned to property names. In fact, property names are actually association lists, to see this association list use the function *Object.getOwnPropertyDescriptor(alist, 'property_name')*, and set these properties of property names use *Object.defineProperty(alist, 'property_name', descriptor),* where *descriptor* is alist that holds the new values.

Yes, in JavaScriptLandese, just like for all foreign languages, there are a many unusual cases, idioms, and exceptions that just have to be memorized if one wants to speak it well. A good dictionary is indispensable.

Arrays

An array is an association list where the property names are integers. Actually, JavaScript acts like such property names are strings where it just happens to be the case that these strings only contain digit characters.

Arrays are given a special read and print syntax by the interpreter, that of square brackets and implied indexes.

```
        [‘a’, ‘b’, ‘c’]  / / an array of 3 strings
```

Figure 17: Reader Syntax for an Array

Although they read and print differently they are still just association lists. Entering the following association list will read to the same value as the example just above:

```
        { 0:'a',  1:'b', 2:'c', length:3,
__proto__:Array.prototype}
```

Figure 18: An Array of Three Strings as an Association List

Here the *length* property is the length of the array. The __*proto*__ property provides the type that this association list is to be interpreted as. This is an array so the type is set to *Array.prototype*. When the printer sees that an association list has a __*proto*__ property equal to *Array.prototype*, it will print using the square bracket notation. However, it might not be a good idea to directly manipulate the __*proto*__ property by name, as it seems that some interpreters perform optimization on this field and might start acting funny afterward. There are functions defined on *Object* for manipulating __*proto*__.

Arrray.prototype is not a literal value, but rather a reference to the predefined *Array.prototype* association list. *Array* is an association list that is predefined as part of JavaScript, and it has a *prototype* property. The *prototype* property of *Array* is an association list containing all the functions that may be used to operate upon *Arrays*, or in other words, it is an abstract type declaration for *Array*. If you enter "*Array.prototype"* into the JavaScript reader, it will respond by printing "*concat*, *constructor*, *copyWithin*, *entries*, *every*, *fill*, *filter*, *find*, *findIndex*, *flat*, *flatMap*, *foreach*, … etc." i.e. a list including the constructor and all of the functions that may be used to operate on arrays. Without *constructor* if we have a reference to *Array.prototype* we will have no way to get back to *Array*, so I assume that is why it is included.

So why do they call it *prototype* instead of 'AbstractTypeDefinition', or 'FunctionList' or some such? You may as well ask why the sun rises. It is just the way it is. Though be careful with this false cognate. The term *protoype* as it is used in JavaScriptLandese refers to a completely different concept than the same term as it is applied to languages like C, where a *prototype* would show the types of the arguments and return values for a function, and would typically be included in a header file. In engineering in general a prototype would be a simplified version or a model of something. There really isn't anything prototypical in this sense about JavaScript prototypes. As a rule of thumb, one can simply drop the word 'proto' from 'prototype' to know how to use the term in Homelandese.

Actually our associative lists are also given a type in this same manner. When you enter an associative list directly, the reader will give it a __*proto*__ property set to Object.prototype:

```
> p = {a:1, b:2}
> p.__proto__ == Object.prototype
 true
```

Figure 19: Associative Lists are Given __proto__ == Object.prototype

Assignment

When a primitive value is assigned to a variable, that variable will receive a *copy* of the primitive value. In contrast, with the default getters and setters, when an association list is assigned to a variable, that variable will receive a *reference* to the association list.

In the example just below we see that the variables *x* and *y* are independent, i.e. that the assignment operation of *y* = *x* copied the contents of *x* and put it in *y*.

```
> let x = 5
> let y = x
> x = x + 1
6
> y
5
```

Figure 20: Copy Assignment

In the following example nothing differs except for the value the first variable is initialized to. However, this time the assignment operation of *b* = *a* puts a reference into *b.* i.e. it does not make a copy.

```
> let a = [1, 2, 3]
> let b = a
> b.push(4)
> a
[1, 2, 3, 4]
```

Figure 21: Reference Assignment

Object.assign can be used to copy the fields of an object, though it will not change the __*proto*__ field of the destination. Hence we can copy an array in this manner:

```
> let x = [0, 1, 2, 3]
(4) [0, 1, 2, 3]
> let y = []  // same as let y = {length:0,  __proto__:Array.prototype}
[]
> Object.assign(y,x)
> y
(4) [0, 1, 2, 3]
> y.push(4)
> x
(4) [0, 1, 2, 3]
```

Figure 22: Copying Array using Object.assign

The array constructor notation is clearer and easier to type, even better is to use a type specific method for doing a copy. Here is an Array copy done the way JavaScriptLanders typically do it:

```
let a = [11, 12, 13]
let b = Array.from(a);  //  or b = […a]
b.push(14);
a
[11, 12, 13]
b
[11, 12, 13, 14]
```

Figure 23: Copying an Array using a Constructor

This makes use of the function *from*, which is found directly in the Array association list because it is a constructor (rather than being found on the prototype list). *from* instantiates a new Array and initializes it from another Array passed in. An alternative reader syntax is shown in the comment. That makes use of the *spread* operator which is three dots. It is considered fashionable to use spread operators whenever possible in JavaScriptLand these days.

Variable passing into functions is a kind of assignment, so now is a good time to ask the question, "Are function arguments passed by value, or are they passed by reference?"   Variables holding primitive values are passed by value. Variables referring to association lists are passed by reference. However there is a simple hedge for passing a primitive value by reference, and that is to put the primitive value into an array of length one, and then to pass that array.

This mixed approach to assignment and argument passing can be confusing, but it is not unprecedented. Our own aboriginal Lisp people have the same custom.

Functions

I have been referring to *functions* during the discussion of association lists, arrays, and assignment. I've been doing this while using the word *function* in the JavaScriptLand sense of the word, while you probably thought I meant it in the sense that we Homelanders are familiar with. You will probably not be surprised at this point to learn that a JavaScriptLand *function* is really an association list that carries information that the interpreter can use in order to find the function code in the source, or even that other things will also be packed into this same association list that in the proximate sense have nothing to do with the function.

A JavaScriptLand *function*:

property name	description
length	number of arguments
name	name of the function, or where appropriate, ‘anonymous’
arguments	upon call holds info about the arguments including their values
caller	info about the caller
prototype	a simple function is an abstract type of its own
__proto__	Function.prototype

Figure 24: Function Properties

When the printer sees the *Function.protype* property it will also output familiar function notation. Similarly, the reader will accept familiar function notation. However this is misleading, as it isn’t a function that we are building. Rather it is an association list with a *__proto__* property set to a reference to *Function.prototype.* In turn *Function.prototype* is a list of *functions*. Now you are sharp programmer, and I hear you asking, “so what good is a list of functions to a function?” Well none really. What a JavaScriptLander calls a “*function”* has the same form as does a class with one method, that one method being the constructor. In Homelanderese such a beast would be called a *functor*. i.e. a singleton class playing the role of a function. Then this *functor* is of type *Function.prototype*. That is why JavaScriptLanders might do things that we would find to be quite quizzical if we did not know this, such as adding a method to what they call a “*function”*.

Classes

As part of diplomatic and friendship effort the government of JavaScriptLand has added syntactical sugar to the reader so that Homelanders may define classes in words they are familiar with, and thus not have to learn to speak the whole of JavaScriptLandese. As you might imagine, a few JavaScriptLanders are unhappy about this, and refuse to use it.

Here is our *Group* class example in the ES6 version of JavaScript:

```
class Group {
        constructor(identity){
                this.identity = identity;
        }
        inverse(a){
                return -a;
        }
        plus(a,b){
                return a + b;
        }
}
```

Figure 25: Javascript Class Declaration

After entering the above on the console, and using *console.dir* to print this, we get:

```
1.class Group
      1.arguments: (...)
      2.caller: (...)
      3.length: 1
      4.name: "Group"
      5.prototype: {constructor: ƒ, inverse: ƒ, plus: ƒ}
      6.__proto__: ƒ ()
      7.[[FunctionLocation]]: VM281:2
      8.[[Scopes]]: Scopes[2]
```

Figure 26: The Associative List for a Class Declaration

So our 'class' has been read into an association list and the *__proto__* property was set to a reference to *Function.prototype*. The printer then followed the *Function.prototype* reference and continued to show us members of that association list. This is interesting, as we can see the location of the constructor in the source code, *[[FunctionLocation]]*. When I click on that link, the IDE opens up a viewer on the constructor source code, and highlights the first line.

Notice the *prototype* property is indeed an association list holding the methods of our class. Hence the functor for the constructor has been given a type definition of our class.

The variable *identity* is missing. That is because non-static non-constructors belong to instances rather than to the class definition. The instance that is allocated by *new* will be passed to the constructor through a scoped variable called *this*. The constructor then creates the property *identity* on the instance. The property *identity* may be created and initialized by other methods, or even by external code.

We could have built this class instead by starting with a functor declaration and then adding our class methods to the functor prototype. Actually, before ES6 this was the most common way to do it. This is also how patriotic JavaScriptLanders still do it, and they like to brag about it. It is only polite to say things such as 'ooh' and 'wow' when they do this.

If I had sat down to design an interpreted object oriented language based on association lists, for a class definition I would have started with three properties for the three parts of a class definition, probably calling them, *constructors*, *instance_format*, and *abstract_type_def*.   I would have used the property name *type* to hold a reference to type definition.   I would set the type of a class definition to something like *ClassDefinition*.

constructors	list of constructor functors
instance format	list of instance properties (no values, this will be copied when instances are made)
abstract_type_def	list of the method functors and static data values
type	ClassDefinition meta type reference

Figure 27: How Tom Would Have Done It

The JavaScript design is a little different, instead of having a property called constructor and listing the constructors, they make the association list itself the first constructor. Thus the association list has an *arguments* property, an arguments *length* property, and a *caller* property, as do other functors. For the *type* property, which is called __*proto*__ they then set that to *Function.prototype* (i.e. type functor). Hence a class definition is indistinguishable from a functor declaration. This isn't too serious of a problem because functors are always classes. However, this unusual organization does have a couple of drawbacks, firstly we miss out in that all of the functors listed on the prototype are for manipulating functors and not for manipulating class definitions. Secondly, there is not an obviously place for holding more constructors, so JavaScript just adds them by name as further properties on the base constructor functor. This is why we find the *from* constructor on as a property of *Array.*

arguments	
length	
caller	
__prototype__	Function.prototype
<some name>	additional constructor
<some name> …	additional constructor
prototype	list of the method functors and static data values
	instance properties implied by assignment

Figure 28: JavaScripts Class Definition

I was a bit confused to discover that the fundamental non-primitive type in JavaScript was an association list, and not just any association list, but one that has special fields for signaling its type to the reader, printer, and thus the interpreter, because so many JavaScriptLanders had told me that the fundamental type was the function. I asked a native JavaScriptLander who teaches JavaScriptLandese to young JavaScriptLanders in high school if he could help alleviate the confusion. Here is our conversation:

Jeramia> The fundamental type in JavaScript is the function.

Thomas> Jeramia, but why is it then that a function has a prototype property, and that the prototype type property has a property that refers to the function?

Jeramia> That is just a wart. You can ignore it.

Thomas> And the prototype property?

Jeramia> Oh that. That is a cancer on the langauge. In the future I think it will go away.

Thomas> *But ...*

Let me hazard a guess as to how we have arrived at this design. I'm guessing that initially the designers said, "everything is a function". But what is a function to a interpreter anyway? It is a structure that holds a reference into the source code where the instructions are found. The language designers must have then chosen to use an associative list for this structure, and it goes on from there. Now we have an object oriented language based on associative lists where these lists are specialized by assigning values to reserved property names.

Perhaps someday I will be able to think like a JavaScriptLander and it will become obvious to me that JavaScriptLandese is based on warty functions. Though had a JavaScriptLander told me that JavaScriptLandese starts with functors, and then these functors can be expanded through the addition of more methods, I just might get that. Though it would help if he or she also said that the functors and their expansions were represented by using association lists.

Having an outer functor being the default constructor, with properties for further constructors, is just a variation in the format of the same information as if we had a constructor property that listed the constructors. While the approach of leaving the properties on the instances allocated via *new* not being declared or defined until run time is a static versus dynamic definition question. Both of these variations are inconsequential. The conceptual structure is the same in Javascript as it is in classical definitions. Here is Figure 7 from the Perspectives on Homeland section shown with the only differences being that the field name *parent* has been changed to the property name *__proto__*, and term *abstract type* has been replaced with *prototype*.

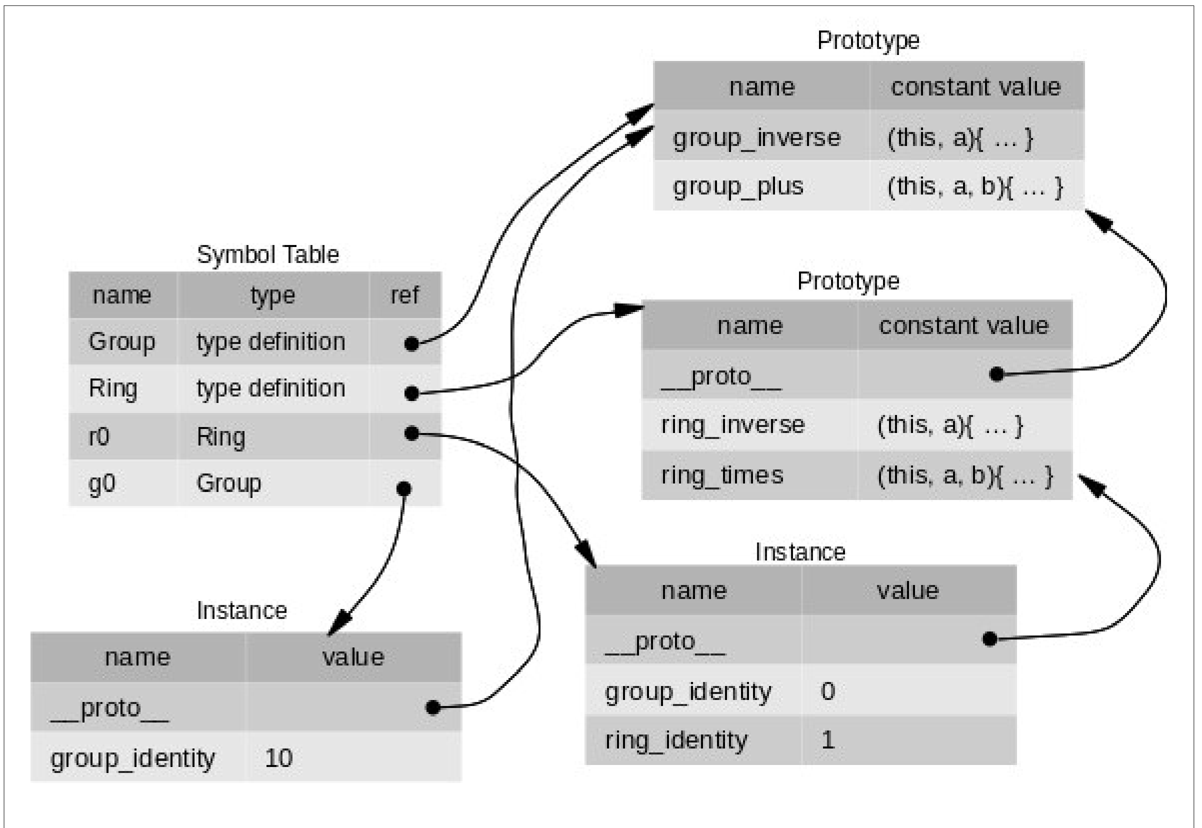


Figure 29: Conventional View with Shortcut References

Going back to our *Group* example, here is the ES6 JavaScript reader syntax for *Ring* which inherits from *Group*:

```
class Ring extends Group {
	constructor(identity, ring_identity){
		super();
		this.ring_identity = identity;
	}
	ring_inverse(a){
		return 1/a;
	}
	ring_times(a,b){
		return a * b;
	}
}
```

After the interpreter reads this and we print it using console.dir, we find the following association list:

```
class Ring

        1.arguments: (...)
        2.caller: (...)
        3.length: 2
        4.name: "Ring"
        5.prototype: Group {constructor: ƒ, ring_inverse: ƒ, ring_times: ƒ}
        6.__proto__: class Group
        7.[[FunctionLocation]]: VM403:2
        8.[[Scopes]]: Scopes[2]
```

As expected this association list has the form of the constructor functor and has been expanded by giving the method functions on the *prototype* property. With just a small nomenclature change to our Figure 7.

The *new* operator will call the *Ring* constructor while passing it the newly allocated instance through a variable called *this*. The Ring constructor calls *super*(), which then runs the parent constructor on the same *this* variable. After super() returns the child will use the same *this* so if any child instance properties have the same name as a parent instance property, the child will overwrite the value the parent provided. If this is not intended, then there is a bug. That is the mechanism behind the JavaScript name space problem.

This problem of potential collisions between parent variables and child variables is exacerbated by the fact that instance properties are dynamically defined. Indeed if one wanted to do a compile time check, there might not even be anything to look at. Anytime the code is changed, there is a new potential for unexpected interactions between parents and children.

You can see this problem in our Group / Ring example by removing the prefix on the *group_identity* and *ring_identity*. The two will then reduce down to the one *identity*. The *Ring* constructor will first set its version of identity to unity, return from super, and then the *Group* constructor will set the same value to its version of zero. Hence, we are forced to have the same value for the additive and multiplicative identity, something that is not likely to work out well.

On stack exchanged I asked an experienced JavaScript programmer what he did to prevent such name collisions. He replied:

> "@user244488 I personally never just extend a class, I always skim through its fields and methods to check whether it makes sense to extend it at all. And if a project gets that large that you won't remember the class you written a year ago, then I always use Typescript (because the IDE spots such mistakes then)"

Well except that Typescipt might not help as it is a static checker. His answer is apparently that JavaScript is only for small projects where the one programmer has control over the whole of the source code, maintains an understanding of how it all works, and can ad hoc rename things up and down a hierarchy as needed.

I have adopted the following convention. In the constructor I assign to *this* all of the instance properties that might ever occur, even if I don't have definitions for them, and I name with a prefix that is the same as the class name. This is how we got *group_identity* and *ring_identity* in the *Group* and *Ring examples.* With this convention there can be no collisions. If a collision is on purpose, one name is used in both places, and that becomes the special case an is labeled accordingly.

A similar issue occurs with method names. Javascript will look for a method name on the child's method list first. If it doesn't find it there, then it will look at the parent. Unlike for the instance variable name collisions, where one variable becomes shared by both the child and parent, both methods will still be there, and will have their original definitions. It is just that the caller can not use a direct call to reach the parent's method. The caller may search through the shortcut links and pull it out. Sometimes we create such collisions on purpose, as we want the child method to take over and thus implement polymorphism. The problem is that the programmer has no way of expressing his or her intentions of what is on the interface and what is not, i.e. when this should happen and when it shouldn't.

Personally I'm 'not uncomfortable' with the possibility of the collision of method names on the interface, because the interface is part of the documentation, i.e. methods constitute the abstract type definition, so I better know what they are. So I don't mangle their names with prefixes. (I *am* concerned about collisions in *this* properties.) However, should there be a helper method that is essentially private, and the programmer was just using it to better structure the code, then such a method better not unintentionally be masked by another method in a child class. Hence, for such helper methods I also prepend the class name followed by an underscore. In our silly Group and Ring examples, then, the listed methods are on the interface, and I would not normally prepend the class name to them. Still however, there will be an unintended consequence, without the prefixes the *inverse()* method of the *Group* would be masked by the *inverse()* method of the *Ring*.

Composition

The Composition pattern for inheritance can be used to solve the parent namespace problem, and it can also handle multiple inheritance, something missing from ES6. Accordingly, the constructor code for the child would call new on the parent classes to make parent instances properties of on the child instance. Here is class field that inherits by composition both Group and Ring.

```
class Field {
      constructor(additive_identity, multiplicative_identity){
          this.Group = new Group(additive_identity);
          this.Ring = new Ring(multiplicative_identity);
      }
}
```

Figure 30: Parent Namespaces and Multiple Inheritance through Composition

When we make a new *Field*, we will get a new *Group* and a new Ring. Hence, there will be no collision on the *identity* instance properties as there was with JavaScript *extend* style inheritance. With this example, to call methods we must provide the parent scope:

```
field0 = new field(0, 1);
let ri = field0.Ring.inverse(5);
let gi = field0.Group.inverse(5);
let product = field0.Ring.times(ri, gi);
let sum = field0.Group.plus(ri,gi);
```

Figure 31: Composition Pattern Method Calls

Thus composition gives us an intuitive syntax for individually calling the otherwise colliding *inverse()* methods. However, we still must provide the long form of method names even when it is not needed, as for *times()* and *plus()* in this example. Perhaps it would be better here to manually add stub methods that call the proper parent methods:

```
class Field {
      constructor(additive_identity, multiplicative_identity){
          this.Group = new Group(additive_identity);
          this.Ring = new Ring(multiplicative_identity);
      }
      additive_inverse(a){ return Group.inverse(a); }
      multiplicative_inverse(a){ return Ring.inverse(a);}
      plus(a,b){ return this.Group.plus(a,b);}
      times(a,b){ return this.Ring.times(a,b);}
}
```

Figure 32: Pulling Methods from the Component Parents Manually

This works well for this example, but it is more common that want a simpler structure where child methods just mask parent methods of the same name, and calls to parent methods made on the child just work. In order to implement we need some sort of continuation structure, where first we look for a method defined for the child, then in not finding that we continue on to look for the method on the parents. Such ability to run continuations are computationally fundamental [1], which is why they are built in to JavaScript through the *__proto__* property and the *extends* syntax.

[1] https://www.researchgate.net/profile/Thomas_Lynch10/publication/315808386_Tom%27s_Turing_Complete_Architecture_Volume_1_Foundation_and_TM_Library_Interface

A user defined continuation mechanism existed in JavaScript at one time. There used to be a __*noSuchMethod*__ property which could be used to provide a function to be called when no method matched [1]. The function attached to __*noSuchMethod*__ could then go check the parents in the composition list for the desired method. Inexplicably, it has been deprecated.

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/noSuchMethod

The documentation says that this feature has been replaced by *proxy* (JavaScriptLandese), which may be used to define an exception handler to catch calls to *get*()[1]. Note this is not specific to method calls, but rather it will affect all *get()*s. I pulled the following example from the Internet:

```
const obj = {};
const proxy = new Proxy(obj, {
  get: () => {
    console.log('inside the get handler');
  }
});

obj.a; // prints "inside the get handler"
```

However www.thecodebarbarian.com reports that a *get()* through a proxy is *a magnitude* slower than a direct *get()*[2].

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy

[2] https://thecodebarbarian.com/thoughts-on-es6-proxies-performance

As a workaround to not having continues in call a Stack Overflow poster suggested scanning the parent classes for the methods, and then to pull them down to child. Taken literally that wouldn’t work, as the copied methods would then be given the wrong value for *this*. We also need to be careful not to burden every single instance with a redundant list of methods. However, it is possible to scan the parent classes and to automatically create stubs in the child class. Then all instances would share one method list, and those methods would receive the correct *this* instance when called. The result reminds me of virtual function tables in C++.

Here the function, *get_method_names*, returns the method function names that constitute an abstract type definition.

```
get_method_names = function(at, no_follow_ats){
  let method_names = [];
  let property_names;
  let descriptor;
  while( at && !no_follow_ats.includes(at) ){
    property_names = Object.getOwnPropertyNames(at);
    property_names.push(...Object.getOwnPropertySymbols(at));
    property_names.forEach
    (name => {
      if (name == 'constructor') return;
      descriptor = Object.getOwnPropertyDescriptor(at, name);
      if (!descriptor || typeof descriptor.value != 'function') return;
      method_names.push (name);
    })
  at = Object.getPrototypeOf( at ); // Object.getParentatOf
  }
  return method_names;
}
```

Figure 33: get_method_names()

And this can be used to automatically inherit parent methods in a child class definition, if that is desired:

```
let inherit_methods = function(child_class, parent_classes) {
  parent_classes.forEach
  (parent_class => {
    get_method_names(parent_class.prototype, [Function.prototype,
Object.prototype]).forEach
    (method_name => {
      if(child_class.prototype[method_name]) return;
      child_class.prototype[method_name] =
        function(...args){ return (this[parent_class.name])[method_name](...args); }
    })})}
```

Figure 34: inherit_methods()

Glossary 1, Useful JavaScriptLandese Phrases

JavaScript Languange	Homelander Language
JavaScript	ECMA script (it has nothing to do with Java)
object	association list
prototype	Built into the language, this is a special association list holding an abstract type definition (a list of functors).
__proto__	Special property because it affects the behavior of the interpreter. References an abstract type.
Object.getPrototypeOf(*obj*)	Doesn't get the *prototype* property of *obj*, rather it gets the *__proto__* property of *obj*. Purposely named this way to confuse Lilliputians and thus prevent them from messing with stuff.
function	functor, i.e. a singleton class with one constructor. Can be expanded to create more complex classes.
clone	copy

Conclusion

Thus, gentle reader, I have given thee a faithful history of my travels for sixteen weeks and above seven hours, twenty one minutes, and 3 seconds on the mark of the period: wherein I have not been so studious of ornament as of truth. I could, perhaps, like others, have astonished thee with strange improbable tales; but I rather chose to relate plain matter of fact, in the simplest manner and style; because my principal design was to inform, and not to amuse thee.