JEEP

A JavaScript Framework for Object Oriented Programming

Design and Development
by
Vinay.M.S

Preface

JEEP is a C++ inspired framework that brings object orientation to JavaScript, well beyond what is available in the language by design. It is an attempt to allow classic software engineering with JavaScript. By that I mean creating reusable, customizable and extensible components of complex structure and behavior. Such tasks are best undertaken with object oriented programming (OOP), but I might be biased in saying that due to my background as a C++ developer.

Jeep has a host of features and they all revolve around structure and semantics. As a result, it promotes writing readable code and is very strict in enforcing a set of rules. It delivers all this with minimum overhead, and sometimes with none at all. Thus, it can be classed as a medium sized framework with moderate amount of complexity. Yet, it is a very flexible framework and quite simple to use. Such a framework cannot be effectively described in a small text file of the kind that online code repositories use. Hence this document.

In this document I will approach things directly. I assume the readers to be intermediate level JavaScript programmers with an idea of OOP. I expect them to know the difference between class and instance, meaning of hierarchy etc. Therefore I won't dwell on many aspects, though I mention and breeze past them to help with the narration.

Although this framework and document are aimed at JavaScript programmers, I constantly compare and contrast Jeep with C++ to give some perspective to fellow C++ developers who have taken up JavaScript for various reasons. I also summarize the significant points in the appendix for easier access. Those with no C++ background may safely ignore these parts. On the flip side, I will often explain the referenced C++ feature in a small sentence to give native JavaScript developers some idea about what it is all about. The C++ veterans are expected to ignore these bits.

I have written this document optimistically. I hope that people will find Jeep useful and adopt it, sooner or later, to an extent that suits them. So I have explained all major points of interest. While not elaborating on everything, there is still 100 pages worth of material here. This might amount to a thin 150 page book. One reason for this size is the generous amount code that is shown; this is after all a document about programming.

The framework bundles a small general purpose testing framework using which Jeep has been tested well. I wouldn't say tested exhaustively as I lack Knuthistic level of intellect and meticulousness, but I will say that it has been tested quite extensively. There are over 400 tests for both failing and passing scenarios. Due to Jeep being feature rich, I have an inkling that some intricate combinations might have escaped scrutiny. So I won't make bold claims.

Jeep started out as a modest function and immediately got incorporated in my code. When I revisited web development in August 2017, I realized that more could be done and took a detour from the application I was building to improve Jeep. What started out as a five day job ended up taking fifty days. In these fifty days straight, I redesigned and rewrote the code from scratch, wrote all the tests, created the test framework itself, wrote the usage examples, wrote the demo code for this document, and finally wrote this document itself. I did often step back and look at what was happening but, due to the incessant work, I might have missed obvious things. Many blunders might have fallen in my blind spot and escaped scrutiny. I would be embarrassed if they exist, but I would gratefully accept and correct them if pointed out.

I fully expect a mixed response to this framework. A lot of things it does goes against a lot of established notions, which are not necessarily correct. Many heads will shake in ridicule and contempt at the first mention of a few features and behaviors. That is fine; I know how we are. But I hope most of those heads fall backwards and ponder a few moments later. I understand that it won't be easy to trust such a non trivial core framework written by an anonymous engineer with no public credentials, and I expect the adoption to be very slow. But the code, concepts and everything else is there for you to consider at your own pace. That was another motivation to write this rather large document. I wanted to discuss things rather than simply mention them. I hope the overall work supersedes personal reputation, or the lack thereof.

I assume and hope that Jeep will appeal to and be appreciated more by C++ veterans who have taken up JavaScript. Approaching with an open mind, native JavaScript developers can also benefit from Jeep, if not appreciate it. Jeep has been central to my JavaScript programming since its inception and has greatly benefited me. I hope you find it useful too.

All the demonstration code shown here is available in an accompanying file for experimentation. Note that Jeep is undergoing refactoring as I write this. One consequence of this is the content of error messages. While the intention, information and the spirit of the messages remain intact, the exact text might change. So this point must be kept in mind if what is shown in this document doesn't match with what is generated during experimentation. As with the implementation code, the test code is also being refactored. So any mention of the number of tests, the names, and even the listed sample code itself must be expected to change.

About Me

I am primarily a C++ developer with almost 14 years experience, the bulk of it as a lead developer and team leader. I have somehow always ended up doing GUI for scientific applications.

I quit my job in December 2012 and have been willingly unemployed since. I wanted to start a company, and almost did, but postponed it. I realized that the ideas and infrastructure could do some more maturing to become a solid business, and there was no reason to rush. I am also not doing a recycled e-commerce or social network thing, so competition is not a problem as well. Further, I am not a fan of vulture capitalists and would use my own savings or bank loan instead. So there is no reason at all to commit to ideas that haven't matured fully. In this period I have developed many C++ components and frameworks to help with my business. I will publish some of them shortly.

I ventured into JavaScript for the first time in March 2015 when I realized some end user products are better packaged as webapps. Between then and now, October 2017, I have a grand total of six months experience with the language spread across the duration. Between September and November 2015, I converted one of my desktop applications into a webapp, and developed an accompanying Firefox extension. I setup a website and uploaded it hoping to make it beta but I didn't do any promotion as I got occupied elsewhere. When I returned, I thought it needed some more polishing and downgraded it to alpha. The work is ongoing.

Though I have created a non trivial webapp, and built a library for UI and other utilities in the process, I still consider myself a novice in the language. Therefore I welcome feedback from seasoned JavaScript developers. I request that the feedback be limited to conceptual and technical issues that I might have got wrong rather than suggestions about coding style etc, not even with "I know you said but..." preface. General technical feedback is also appreciated.

I am from Bengaluru, India. Apart from writing software, I have a lot of interests in diverse fields. It just happens that writing software can bring me the financial backing necessary to be able to pursue others, now and later. So I am a professional software engineer. I would describe my personality in two words – ambitious (within ethical limits) and autodidactic.

You may contact me at engineer.vinayms@gmail.com

Contents

1. Introduction

JEEP is a JavaScript framework that brings object oriented programming (OOP) facilities to the language. There are already many such frameworks in existence, so you might question the necessity of yet another framework on those lines. The answer to that lies in a small personal story.

1.1. Motivation

I am primarily a C++ developer with more than a dozen years of experience, but I am by no means an expert or a language lawyer. Neither have I amassed hundreds of thousands of reputation points on online forums. I just use it to build things. I have built a variety of things in C++ and am really fond of the good parts of the language. I missed them badly when I was forced to consider JavaScript in mid 2015. Briefly, I wanted to solve the portability problem of one of my desktop applications. I thought converting it into a webapp will help but little did I expect web development, especially front end, to be the quicksand that it is.

When I wanted to incorporate OOP techniques in my JavaScript code, I researched for frameworks that made it possible but none of them that I found looked appealing. They were all either ad-hoc, incomplete, abandoned, lacking, bloated or not even remotely OOP related. So, as any software engineer would do, I created one for myself. The resulting framework that is Jeep goes far beyond existing frameworks in terms of usability and feature set. Therefore, I won't hesitate to say that it is quite necessary. Jeep is the result of my trying to adapt JavaScript to make me feel at home.

To be clear, Jeep is an attempt to make JavaScript code look, feel and behave like C++ code.

This might make some of you shudder. For the uninitiated, both JavaScript and C++ aren't known for their robustness. They are quite fragile languages, and in the hands of an incompetent programmer they wreak havoc. There are valid reasons for the fragility. It is evident that C++ had an enormous corporate pressure and practical constraints during its evolution, while JavaScript was created over a weekend. Combining two such languages might seem particularly repulsive. On the flip side, competent programmers understand the limitations of these languages and abide by rules and conventions. As a result, they reap all the benefits offered by these languages. It would be a mistake to ignore the power of these languages. It could be argued that one's experience with these languages is reflective of one's competence.

Jeep tries to bring the best of C++ to JavaScript and in the process bring some structure and discipline to the unruly JavaScript code. This fact is also the source of its name (js cpp -> j seep -> jeep). The aim is to elevate JavaScript from a language that does DOM manipulation using scattered functions, to a language suitable for proper software engineering. These are the days when JavaScript is being used on server side, and even for building desktop applications. These are the places where software engineering is absolutely necessary. Yet, not much seems to be done in that regard. It seems that the only developments in the world of JavaScript are improvements of the engines, or yet another mega corp promoting yet another half baked one page application framework that is the 23rd clone of something that was created ten months ago. There seems nothing done at conceptual or structural level. Jeep tries to fill that gap.

Among the "traditional" programming languages, C had the reputation of being particularly nasty compared to its peers like Pascal, or even by itself (I don't know how it is today, I don't use it). It demanded very high level of programmer competence to use it; but offered mighty rewards in return. Then came the C++ language. While the explicitly stated goal of C++ was to allow better organization of programs, as computation was already handled rather well by C, it can be interpreted as bringing some order to the unruliness of C. The interpretation proves correct when reading the design motivations of C++. I consider Jeep something like that. If not C++ itself, Jeep is at least C with Classes, the very early version of C++. What it did to C, for better or for worse, Jeep does to JavaScript.

Closer inspection will reveal more parallels. All the reasons why C++ chose C as the base and not something like Pascal is somewhat relatable with Jeep trying to fortify JavaScript. While C++ tried to make C more like Simula, Jeep tries to make JavaScript more like C++. The design goals of C++ and Jeep are also quite similar – trying to provide all the features with minimum overhead both in terms of space and speed. I am not equating either the designers or their products, but merely pointing out the many parallels they have. I find this quite intriguing. Obviously, I did not plan on doing anything to this effect; it just happened. I realized this only while writing this document

1.2. Features

At the outset I must make it clear that Jeep is a pure JavaScript framework without any relation to DOM or any other web technologies. It deals with the JavaScript language only and not the environment in which the language is used. Jeep is an independent framework - it is not built using or extending any other framework. It written in standard JavaScript (ES5), so it should work on all standards complaint browsers, probably even on Internet Explorer.

JavaScript is only vaguely object oriented and it is quite inhibiting for someone coming from a stronger object oriented background like C++. Specifically, JavaScript doesn't offer multiple inheritance and virtual function mechanism (including pure virtual functions), both of which are vital to developing non trivial applications based on OOP principles. These features were the original motivation behind creating Jeep.

As mentioned, the OOP features provided by Jeep are inspired by C++ and go well beyond what is natively possible with JavaScript. Most behavior is retained from C++, but there are some changes owing to the nature of the JavaScript language. Some other changes are in those aspects of C++ that I don't like as they seem conceptually unsound. I took this opportunity to design them in a way that I feel is conceptually sound. It might come across to you as quite amusing that an anonymous engineer has such strong opinions that challenge the collective intelligence of a whole committee of accomplished computer scientists and engineers who designed C++. I agree.

Jeep has far too many features to list. Jeep is also quite intricate in things it does. If named, many of the features end up being long phrases; some would be clauses. It is hard to express succinctly even in a language like English that routinely poaches words unabashedly and gets butchered on a daily basis to create neologisms. Therefore I oversimplify Jeep and reduce it to the following lists.

Qualitative

- robustness
- maintainable and extensible classes
- intuitive class description
- improved productivity and performance

Technical

- multiple inheritance
- virtual and abstract functions
- public, protected and private members
- development and production mode split

Jeep works on the *pay per use* principle. Many of the features, mainly the ones that are too alien to JavaScript, need a bit of work to get to work. Consequently, they have some overhead in terms of space and performance. But these features, like all features, are explicitly availed by the programmer. So, as long as costly features are not utilized, you can expect near native performance, the performance you get with plain JavaScript code. I qualify it with *near* because Jeep is still gong to be a couple of milliseconds slower. It might not make difference qualitatively, but quantitatively it is slower indeed. Even when costly features are utilized, as long as they are not used in performance critical points of the code, the tens of milliseconds overhead that they have would be hardly noticeable in the overall scheme of things.

1.3. The Nature of Jeep

Although Jeep is referred to as a framework, it is more than that. It is also a programming language and a development environment. Jeep is three things at the same time.

- It qualifies as a framework due to its fundamental nature. As frameworks do, Jeep gives the programmers slots to plug in their code while doing all the heavy lifting behind the scenes. It also offers an abstraction layer that helps in describing things.

- It qualifies as a programming language because it has syntax, semantics, and imposes many rules to keep things consistent and logical. Essentially, Jeep code is compiled to plain JavaScript code (so may be it is a compiler too). During the compilation process, and even run time, specific errors are issued whenever the language rules are broken.

- It qualifies as a development environment because it offers separate modes of working tuned to different purposes. The two modes, called development mode and production mode, are tuned towards offering the developers maximum debugging assistance and the users maximum performance respectively. There are also several flags that control many aspects of the environment.

There have been several attempts made to tame the feral language that is JavaScript. One popular solution is to write code in some other scripting language and then compile it into JavaScript. This language could have varying levels of mutual intelligibility with JavaScript. This solution is the industry standard to all programming language problems – invent another programming language in which to code and then transcompile to the original language. While it is an interesting project for language designers and compiler writers, it is a poor solution to the problem. We have to wrestle with two languages now. Aren't we swamped enough already? In contrast, though Jeep looks and feels like a different language, it is still JavaScript. There is no need to learn yet another evolving and soon to be abandoned language with an oddball name. You can start using Jeep directly. The name is oddball enough.

To be clear, Jeep is a JavaScript processor written in JavaScript to process JavaScript code and produce JavaScript code.

This nature of Jeep obviously has some performance issues. It would be foolish to deny that Jeep will be slower compared to plain JavaScript. Jeep will always be slower than plain JavaScript no matter what kind of ninja code is used in implementation, but what is important is whether the slowness affects qualitatively. This will be discussed in the ***Internals*** chapter.

The two modes of working are present precisely to remedy this situation. Those experienced with development with C++ are quite aware of the benefits of such a separation between development mode and production mode (usually called build and release mode respectively). Jeep brings this aspect to JavaScript, while still being JavaScript.

As a language, Jeep provides both compile time and runtime robustness. Compile time robustness is basically enforcing language rules and refusing to compile erroneous code. For example, something that is declared as a variable but assigned to a function. Error reporting was one among the bad features of C++ for a long time, and still persists. While technically a compiler issue, it is attributed to the language itself. That experience has made me make Jeep issue precise errors, as much as possible, detailing exactly what is causing the error. The runtime robustness consists both OOP related scenarios and general scenarios. For instance, not instantiating a class that has abstract functions is an OOP related scenario, while not running a function invoked with wrong number of arguments is a general scenario. All robustness features are controlled at three levels – class description, environment mode, and environment flags. Due to the nature of JavaScript, both compile time and runtime errors abort the script.

Further, Jeep is bootstrapped, or dogfooded if you will. The implementation uses some low level OOPish structures that are made available as the part of the framework. To be honest, these things were late additions to the framework done mid way. They became necessary while trying to maintain code reusability and my sanity. When done, they seemed like a generally useful set of things, so they are part of the framework itself. The fact that such things needed to be created is a testament to the fact that a Jeep like framework is quite necessary. This dogfooding has an interesting implication. All the code must remain in the same file. Thus, Jeep is a one big file of about four thousand lines of code. But mind you, the code is written with a lot of breathing room and a generous amount of comments. It uses mostly C++ style scope braces. So it again proves how useless the LOC metric is.

2. Overview

The framework is defined inside the `JEEP` object in the *jeep.js* file.

```
JEEP = {
    CreateEnvironment: function(info){},
    GetRecord: function(name){},
    GetStructure: function(name){},
    GetClass: function(name){},
    SetStdErr: function(printer){},
    Utils: {},
    impl: {},
}
```

The *impl* object contains all the implementation, and is not supposed to be touched. It is expected that this programmer's contract is upheld. The *Utils* object contains some utilities that might be useful to the Jeep's clients; the utilities are used in Jeep's own implementation. The *SetStdErr* sets the location where the error gets printed. The argument is supposed to be a function taking one argument and printing it as is fit. By default, errors are printed to the console. The demo code uses this function to redirect the errors to the webpage.

2.1. Objects

There are three kinds of objects in Jeep – records, structures and classes. Records are the simplest ones and classes are the most complex; structures are in between. Note that C++ doesn't have records, and the name is taken from Pascal. The *Get* functions retrieve the object definitions that are stored in a common database.

The reason for using the database is two fold. One, it protects from accidental overwriting of the definition objects. These objects generally tend to be in global scope and are at the risk of being reused by an overly enthusiastic programmer who also has inconsistent variable naming conventions. When that happens, the definition is lost. Of course, the script won't run in such scenario, but using a database saves the hassle in the first place. It is reasonable to assume that the result of the *Get* functions would be stored in a local scope and less prone to careless reuse. The second reason is sharability. The database is common and available globally to all code on a webpage (or in any other context). This makes sharing and using the object definitions simpler. When objects are part of a library, this avoids juggling many global definition objects and mismanaging them. The database is deep inside the framework, so nothing short of deliberate mischief can upset the setup.

It's a bit ironic that despite such strong sentiments about global objects, the *JEEP* object itself is global and open to all the problems listed. There could have been a *GetJEEP* function, but what if that function is overwritten? As they say, its turtles all the way down. The database offers a simple but potent solution to a not so uncommon problem. As for *JEEP* getting overwritten, the only way to solve it is to set a precedent by firing the first person who does it.

The objects must be first generated before they can be instantiated. Remember, since Jeep is also a language, the JavaScript code must be generated from the "Jeep code" first. There are two kinds of generation methods - one that returns the definition object and one that doesn't return but adds it to the database. The first method exists to allow creation of local objects that need not be part of the global database. All this happens within an environment.

2.2. Environment

The most important function is the `CreateEnvironment`. Based on the parameters given, this function creates an environment that is targeted to either development or production. This is akin to build settings that C++ systems have.

```
AppEnv = JEEP.CreateEnvironment({mode: “development-mode”})
AppEnv = JEEP.CreateEnvironment({mode: “production-mode})
```

The development mode is slower because it does a lot of validation while the production mode is optimized for speed and skips all validation, unless specified by the additional flags. The flags control how the code is generated. The environment object that this function returns contains the real Jeep API.

```
Environment = {
    Object: {},
    Function: {},
    IsDevMode: function(){},
    CreateNamespace: function(){name){},
}
```

The API is divided into two groups, one for objects and one for functions. The function API present in `Function` exist only to bring many class member function features to plain JavaScript functions. To understand them, class functions must be understood first.

The environment object contains a utility function `IsDevMode` that returns true if the environment is in development mode. This result can be used by clients of Jeep to do their own implementation variation. The general usage pattern with Jeep is to develop in the development mode first and then use the production mode. Switching is literally a matter of using two slashes to comment out the other mode.

The *Object* contains the API for generating object definition. Since there are three types of objects and two methods of generation, a total of six generating functions exist. All functions have the same syntax of the following form. The class being the most complex has a variation to the registration process which will be discussed in §4.2.

```
CreateX(name, description}
RegisterX(name, description}
```

The name is a string. It must be alphanumeric and underscore only, with an alphabet being the first character. For simplicity, alphabets are limited to the English language. The dollar symbol is disallowed because it is a special marker that is central to implementing all of Jeep’s features. The description is an object with several properties. This argument remains mostly consistent in all functions but contains more properties for more complex objects. The `Create` function returns the definition and the `Register` function adds it to the database. The `Get` functions mentioned earlier use the same name as used to register. The name rules and properties and their values essentially constitute Jeep’s syntax.

There is no way to delete definitions from the database because it is absurd from Jeep’s point of view. However, the same effect of registering, accessing and deleting can be achieved by simply creating local object definitions with `Create` functions.

2.3. Namespace

Since the database is common, there can be name clashes. As expected, attempting to register more than once with the same name is an error, and it aborts the script. The solution to this is using namespaces. A namespace groups objects such that the names don't clash while still using the common database. Namespaces are created with the `CreateNamespace` function of the environment object. The function takes a name as its parameter. The name follows the same naming rules as object's, except that valid names can be separated by a single dot to create sub namespaces directly (a more structured way is described in §8.2.1). The function returns an object that has the Object API only because Function API are not name centric

When the API functions present in the namespace object are used, object generation and access are limited to the local group setup by the namespace. In contrast, using API in the environment object operates on the global scope. The namespace grouping mechanism is quite simple – the namespace name is prefixed to the object's name. So, while the generators and `Get` functions of the environment object can access the objects created by namespace by typing out the fully qualified object name, which means prefixing the relevant namespace names to the object name, using a namespace object is simpler and less error prone. Any number of namespaces can be created and it has no bearing on the database unless the API functions are invoked. There is no distinction between opening and creating a namespace.

Namespaces help building libraries. The general setup for libraries would look like this.

```
// libx.js
function InitLibraryX(Env, libparams)
{
    Lib = Env.CreateNamespace("LibX")
    Lib.RegisterClass(...)
    return Lib
}

// client.js
AppEnv = JEEP.CreateEnvironment(...)
LibX = InitLibraryX(AppEnv, {...})
C = LibX.GetClass(...)
```

3. Record and Structure

As mentioned in the overview, records and structures are simpler classes. They are simpler in syntax, structure, semantics, behavior and function. The syntax particularly is a subset of the class syntax. Therefore, understanding them helps understanding classes. This chapter is an essential reading for the chapters about classes. Further, records being simpler than structures, understanding them is will help understanding structures.

3.1. Record

A record is just a named collection of variables. The description might make it seem useless, but a record is a lifesaver for a language like JavaScript. Records are the building blocks. They also improve performance, though that fact is not obvious.

Consider this code for some kind of file management.

```
let Flags = {F_COPY: 1, F_COPYERASE: 2, F_RAIDCOPY: 4}
let actions = [];

function event1(){
    //...
    actions.push({src: "c:\\afile", dest: "d:\\file_a", flag: Flags.F_ COPYERASE})
}
function event2(){
    //...
    actions.push({src: "d:\\file_x", dest: “[home]", flag: Flags.F_RAIDCOPY})
}
function event3(){
    //...
    actions.push({src: ""[office]file_x ", dest: “[home]", flag: Flags.F_COPY})
}
```

It is your typical JavaScript code. You might be so used to it that unless you step back and think, or you are an immigrant from a different language like C++, you might not see the code bloat here. You might also not see how cumbersome and error prone it is to repeat similar code in multiple places. The point about having to repeat, particularly, is against the core principle of software engineering and undesirable. Records are meant to solve these problems.

Consider the same code with records.

```
let Flags = {F_COPY: 1, F_COPYERASE: 2, F_RAIDCOPY: 4}
let actions = [];

DemoEnv.Object.RegisterRecord("FileTransferInfo", {
    src: "",
    dest: "[home]",
    flags: Flags.F_COPY,
})

function event1(){
    //...
    let FTI = JEEP.GetRecord("FileTransferInfo");
    actions.push(FTI.New({
        src: "c:\\afile",
```

```
        dest: "d:\\file_a",
        flags: Flags.F_COPYERASE
    }))
}
function event2(){
    //...
    let FTI = JEEP.GetRecord("FileTransferInfo");
    actions.push(FTI.New({
        src: "d:\\file_x",
        flags: Flags.F_RAIDCOPY
    }))
}
function event3(){
    //...
    let FTI = JEEP.GetRecord("FileTransferInfo");
    actions.push(FTI.New({src: "[office]file_x"}));
}
```

Since a record is just a named collection of variables, the description object consists only of objects; having functions is an error. The values set to the variables are the default values the record has. They can be reset by giving an object with a subset of the description object during instantiation (remember, a set is a subset of itself). Every record definition automatically gets the `New` function for instantiation; the `new` operator won't work.

I hope you can see how efficient and productive it is with records compared to plain JavaScript method on several counts. With records, you need not provide all the variables but allow defaults to take over, which is very unlike what can be done with plain JavaScript. This not only guarantees a consistent data structure, it also reduces code bloat (only someone who doesn't understand what code bloat is can argue that the `GetRecord` is code bloat). Using records saves a lot of checking back and forth to see what kind of properties does the object need to have. Without records, you would check other places where such entries are added. If the intended data structure happens to have optional properties that can be left out, then you can never be sure what things are to be given. You can refer to the documentation when it exists, but we all know what are the chances of that. In contrast, the only place you need to refer to with records is the record description itself. The code becomes self documenting.

Further, while instantiating, giving a variable name that is not part of the description causes runtime error and aborts the script. A typo won't accidentally create a new variable and change the shape of the object. This is part of the robustness provided by Jeep. The constant shape has a positive effect on the performance as the engines can optimize freely.

Records are pure data and perform no action. They are meant to be used as map values, array entries etc. Even if the entries are added in one loop and not scattered across various functions, records are useful for the very same reasons.

Like the `New` function mentioned, the framework automatically adds some functions and variables to every record definition and instance. They help in meta processing the records.

Definitions get a variable called `$name` which contains the name of the record. It is the same name given while generating the definition. Such dollar prefixed variables are called *reflective properties*. They are semantically constants. The constancy is enforced via programmer's contract only. These properties are meant to be used by definitions and instances internally. But because records don't have member functions, external code could use them without any guilt.

Definitions also get the function *InstanceOf* that helps determine if a particular object is an instance of a particular record. It takes one argument which it tests.

The instances get *Clone* and *Equal* functions. As the names imply, they help in duplicating the instance and testing the equality respectively. The *Clone* operates recursively on all the variables. It produces a true clone instead of a shallow copy, so a change in a nested object in the clone won't reflect in the instance cloned. The function takes one argument that it clones. The *Equal* does something similar and takes one argument. Testing equality is a two step process. It first tests if the given object is of the same type. For that it simply uses the *InstanceOf* function. Then, it recursively tests every variable to determine equality. Due to the recursion, the performance of these two functions is affected by the layout of the record.

An instance also gets the *Change* function that is meant to update the values of the instance. You could change the variables individually as with plain JavaScript object, but using this function has two advantages - it allows you to change in one line of code (or one function call rather) and it validates that correct variables are mentioned, as with instantiation.

```
let fi = actions[0];

cout("FileTransferInfo.InstanceOf(fi)? "
                              +(FileTransferInfo.InstanceOf(fi)?"yes":"no"))
cout("FileTransferInfo.InstanceOf({})? "
                              +(FileTransferInfo.InstanceOf({})?"yes":"no"))

cout("> cloning")
let ficlone = fi.Clone()
let cloningSucceeded = FileTransferInfo.InstanceOf(ficlone) && fi.Equal(ficlone);
cout("cloning "+(cloningSucceeded?"succeeded":"failed"))

cout("fi.Equal(ficlone)? "+(fi.Equal(ficlone)?"yes":"no"))
cout("fi.Equal()? "+(fi.Equal()?"yes":"no"))
cout("fi.Equal(null)? "+(fi.Equal(null)?"yes":"no"))
cout("fi.Equal({})? "+(fi.Equal({})?"yes":"no"))

cout("> using fi.Change({flags: Flags.F_RAIDCOPY})")
fi.Change({flags: Flags.F_RAIDCOPY})
cout("fi.Equal(ficlone)? "+(fi.Equal(ficlone)?"yes":"no"))

cout("> instantiating with invalid variable")
try{FileTransferInfo.New({type: Flags.F_RAIDCOPY})}catch(e){cout(e.message)}
```

```
FileTransferInfo.InstanceOf(fi)? yes
FileTransferInfo.InstanceOf({})? no
> cloning
cloning succeeded
fi.Equal(ficlone)? yes
fi.Equal()? no
fi.Equal(null)? no
fi.Equal({})? no
> using fi.Change({flags: Flags.F_RAIDCOPY})
fi.Equal(ficlone)? no
> instantiating with invalid variable
JEEP run time error [record FileTransferInfo]. Attempted to initiate with non
existant variable 'type'.
JEEP aborted.
```

3.2. Structure

A structure is closer to a class than a record. A structure has functions and performs actions unlike records that are pure data. The example below shows a lot of details that are very similar to what follows next about classes, so it helps if you study it for a while. It describes a simple numeric array processor called *Vector*.

```
DemoEnv.Object.RegisterStruct("Vector", {
    CONSTRUCTOR: function(rawArr){
        this.arr = Array.from(rawArr);
    },
    Variables: {arr: []},
    Functions: {
        Print: function(){cout(this.$name +": "+ this.arr.join())},
        Size: function(){return this.arr.length},
        ApplyOperator: function(op, vec){
            if(vec && !this.$def.InstanceOf(vec))
                throw new Error("Vector.ApplyOperator expects a Vector.")
            vec = vec || this;
            let res = [];
            for(let k = 0; k<vec.Size(); k++)
            {
                switch(op)
                {
                    case '+': res[k] = vec.arr[k] + this.arr[k]; break;
                    case '-': res[k] = vec.arr[k] - this.arr[k]; break;
                    case '*': res[k] = vec.arr[k] * this.arr[k]; break;
                    case '/': res[k] = vec.arr[k] / this.arr[k]; break;
                }
            }
            return this.$def.New(res);
        }
    }
});

let Vector = JEEP.GetStruct("Vector");
let vec = Vector.New([1,2,3,4,5]);
let res = vec.ApplyOperator('*');
res = vec.ApplyOperator('*', res);
res.Print();
```

```
Vector: 1,8,27,64,125
```

The description object is sectioned into several parts, each mentioning a specific aspect of the structure. This is in striking contrast to the description object of records but very similar to what is given for classes. All the properties of the description object must be self explanatory.

The functions and variables have member status. `Variables` must contain objects only and `Functions` must contain functions only; violating this results in errors. As with records, structures are instantiated with the `New` function rather than the `new` operator. The reflective property `$def` contains the definition of the structure itself. With this, an instance of a structure can create objects of its own kind without having to use the definition object like external code do. This frees the instances from being dependent on a vulnerable global variable to access their own definition as normal JavaScript code have to.

Apart from the auto generated members mentioned, structures get all of them that records get. The functions have identical behavior as well, but enhanced to suit structures. Since structures have functions too, the `Clone` copies functions of the original to the resulting clone. Note that

it copies and not clones functions. In OOP, code remain same but data changes, so copying the functions is sufficient. Cloning would require using the dreaded `eval` and Jeep doesn't touch that even with a radiation suit on.

Constructors are special functions. The name and concept are taken directly from C++. It is the first function called after instantiation. However, having constructors is optional. They are called only if they are defined. Constructors often exist just for the sake of initializing the member variables with values. If that is the only reason to have a constructor in a structure, there is a better solution to reduce the code bloat.

Structure definitions get `InitNew` function that helps with initializing variables during instantiation. This function is similar to `New` of record in all respects. Constructing an instance with this function is called *init construction*. With init construction, it is not necessary to define the constructor. But if defined, constructors are called with the instance already initialized. When called during init construction, constructors get no arguments.

Init construction can be done in another way. When the `New` function is called with the text "*init*" as the first argument and an object as the second, it behave exactly like the `InitNew`. For other combinations, the constructor is called as usual. This variation exists to give some flexibility in designing the construction mechanism. Instead of using a separate function, just adding or removing the "init" text transforms the construction mechanism. It also exists because classes have them, and it seemed like good idea to mimic it here.

```
let Person = DemoEnv.Object.CreateStruct("Person", {
    CONSTRUCTOR: function(){
        cout("Constructor args: "+ JSON.stringify(arguments)+
                                    " fullname: "+this.GetFullName())
    },
    Variables: {
        firstname: "",
        lastname: "",
    },
    Functions: {
        GetFullName: function(){
            return this.firstname + " " + this.lastname
        }
    }
})

cout('> using Person.New("John", "Doe")')
Person.New("John", "Doe")

cout('> using Person.InitNew({firstname: "John", lastname: "Smith"})')
Person.InitNew({firstname: "John", lastname: "Smith"})

cout('> using Person.New("init", {firstname: "Shri", lastname: "Samanya"})')
Person.New("init", {firstname: "Shri", lastname: "Samanya"})
```

```
> using Person.New("John", "Doe")
Constructor args: {"0":"John","1":"Doe"} fullname:

> using Person.InitNew({firstname: "John", lastname: "Smith"})
Constructor args: {} fullname: John Smith

> using Person.New("init", {firstname: "Shri", lastname: "Samanya"})
Constructor args: {} fullname: Shri Samanya
```

3.3. Object Mutilation

It is pertinent at this point to discuss about the idea of *object mutilation*. Records, structures and classes, all are susceptible to this. As the complexity of the object increases, so does the side effect of object mutilation.

Adding variables and functions to object definitions outside the generation process, and to instances outside instantiation process, is considered mutilation. It is the result of the looseness of JavaScript. Mutilation could happen externally or internally. Consider this code.

```
let Person = DemoEnv.Object.CreateRecord("Person", {
    firstname: "John",
    lastname: "Doe",
    age: 30,
})
let p = Person.New();
p.city = "unknown"

Person = DemoEnv.Object.CreateStruct("Person", {
    CONSTRUCTOR: function(){
        this.city = "unknown";
    },
    Variables: {
        name: "John doe",
        age: 30,
    },
})
Person.New().Print();
```

The record is mutilated externally and the structure is mutilated internally. The example shows adding only variables, but functions can be added too. Adding members via mutilation undermines the benefit of Jeep and sort of negates its reason to exist. The whole point of Jeep is to provide structural, syntactic and semantic robustness necessary for software engineering while working with a loose language like JavaScript. For this, there is a lot of validation happening during the generation process. And object mutilation promptly undermines all of it.

Object mutilation has a whole gamut of bad consequences.

- It makes object definitions untrustworthy. You can never be sure that what you sees is what exists and must read all the implementation details to be sure.
- It affects performance as the object shape is changed, which hinders optimizations done by the engine.
- It leads to undefined behavior for classes which undergo a variety of validations during generation.

Object mutilation can have dire consequences because it use the backdoor to get in. For a language like JavaScript the backdoor can't be shut; actually it is not really about the door as much as the whole back wall missing.

There is nothing redeeming about object mutilation; hence such an overwhelmingly negative name given to the activity. Note that there is no way to stop mutilation form happening (unless recursively freezing all objects). It comes down to your discipline and desire to write readable, maintainable and well performing code.

4. Class

This chapter builds on the information discussed in the previous chapter, so having read the previous chapter is essential before moving forward. Unlike the demonstration code in that chapter, for simplicity, I will only show dummy code and not attempt to forge any real world like examples for classes. Classes are intricate in structure and behavior, so simple examples help focus on aspects better. Almost every example shows a lot of new detail, so it helps if you consider them for a while.

4.1. Overview

Classes retain constructors, variables and functions introduced in §3.2 but extends them all significantly. Unlike structures classes are instantiated with the `new` operator.

Classes have three privilege levels of access – public, protected and private – as opposed to structures where everything is public. The members added via `Variables` and `Functions` properties have public access. Constructors are always public. The protected and private members are added via `Protected` and `Private` properties respectively. The protected and private members are supposed to be implementation detail, so they are not accessible from external code the way public members are.

Unlike public members, there is no separation of functions and variables for protected and private members. The functions and variables are clubbed in the same property. The effect of this is that if a name was intended to be a function but was instead assigned to a string, it becomes a variable. The programmer must declare things carefully. This choice was made for two reasons. First, it keeps things simpler, from both usage and implementation points of view. Then, protected and private members being implementation detail, it seemed acceptable to have it this way. Further, providing a separation would result in nested properties, which I don't like, or separate properties with long names, which I don't like either.

Classes can also have static members. They are the ones that are not linked to the class instance and can be accessed separately. They are declared inside the `Static` property. Like protected and private members, they too are clubbed for the same reasons. Moreover, static members are going to be used sparingly, so it seemed acceptable to club.

Classes being complex have a lot of features. It is impractical to have them all in every class generated. So, classes allow flexible code generation. The flexibility is availed via flags and directives. The flags are given as a comma separated list of hyphenated words in the `Flags` property of the descriptor.

Directives are applicable to only those functions and variables that use them. With some restrictions, the same directives are applicable to public, protected, private and static members and produce same behavior. In almost all programming languages, directives are keywords added before or after the identifiers and separated by spaces. But because Jeep is really JavaScript, and cannot reinvent syntax out of thin air, Jeep improvises.

In Jeep, directives are part of the name of the members. It is essentially a prefix that decorates the name, but it is separated during processing. It looks something like this.

```
$directive1_directive2$__functionName: function(){}
```

The format is quite simple: directives are confined within two dollar characters, and separated from the name with underscores. The underscores aren't necessary for processing, but enforced to improve readability. The name can be separated with as many underscores as necessary to make the code readable (two with fixed sized fonts should suffice most cases). When multiple directives are necessary, they are separated with a single underscore, but they all still remain inside the dollar characters. As mentioned, the actual name would be separated during processing, so the members with decorated names must be referred with the undecorated names only.

Some flags have a corresponding function directive, but act as a shortcut and apply the directive to all functions in the class. This saves a lot of typing. It is also beneficial when new functions added are to have the same directives. If a function uses a directive that is applied by a flag, there is no problem; it is idempotent. Constructors cannot use directives.

4.2. Generation

There are two methods of class generation - the direct method and the split method - that help manage the amount of implementation details exposed upfront. The direct method is a one step method and hence all the implementation details are exposed. Using the `CreateClass` or `RegisterClass` functions constitute direct method.

The split method is inspired by the C++ way of organizing code. There is an emphasis on separation of interface from implementation. This is done either by having them in separate files altogether (called dot etch and dot sea pea pea files respectively), or in the same file. One of the benefits of this method is information hiding. Doing this greatly helps with reading and understanding the intent of a class as there are no implementation details to distract. It is sort of a Zen mode (just throwing in a much loved word by all and sundry).

The split method brings that benefit, in whatever little capacity, to JavaScript code. It is a two step method and uses two functions, `DeclareClass` and `DefineClass`. This method always registers the definition.

```
DemoEnv.Object.DeclareClass("Class", {
    CONSTRUCTOR: function(v){},
    Functions: {$const$__Print: function(){}},
});

DemoEnv.Object.DefineClass("Class", {
    CONSTRUCTOR: function(v){
        this.value = v;
        cout(this.$name+" instantiated with "+this.value);
    },
    Variables: {value: 100},
    Functions: {
        $const$__Print: function(){
            cout("The value given was " + this.value);
        }
    },
});

let Class = JEEP.GetClass("Class")
let c = new Class(100);
c.Print();
```

The key thing about this method is that anything that is considered implementation detail is disallowed from being declared. Hiding implementation details is not an option, but rather it

is enforced. Private information is obviously implementation detail. So are variables and flags. For protected members, only virtual and abstract functions are allowed because these are legitimate interfaces as well. Note that the declared functions contain no code, and this is enforced too. Functions must be truly empty, with nothing in the block but spaces; not even a semicolon. If the function spans lines it is considered non empty and will flag error. Many function directives are considered implementation detail, and hence disallowed. The `const` directive shown here is a semantic guarantee and legitimate part of the interface, so it and other such directives are allowed.

This method enforces a lot of rules in order to maintain consistency between the code that is split. All directives used in declared functions must be used in define (though order doesn't matter). All defined functions must have the same directives as declared, except those that are considered implementation detail. All the functions in both places must have the same number of arguments. A declared function must be defined, but a defined function need not be declared. The missing ones are considered implementation detail, which is the whole point of this method. Protected functions must remain protected and public must remain public in both declaration and definition.

I hope you see how advantageous it is to have this separation. Even a throwaway example benefits from it. This should be the method of choice for generating classes that are exported from a library, with all declarations shown near the beginning of the library. The split method basically creates a self documenting code, especially when functions are named thoughtfully. A couple of lines of comments about each function's purpose and parameters is all that takes to have a well documented class.

Without this method, the only way to document code is to either have generous comments in the source file, or have a separate document. Whatever the choice made, both the code and the documentation must be kept in sync all the time. In contrast, with this method you don't have to do anything extra. Further, if there is a discrepancy between what the document claims and what the interface does, there will be a problem. Especially with things like argument counts, things can fail silently for a long time. In contrast, with this method if a lazy programmer forgets to keep the changes in sync across declaration and definition, Jeep doesn't generate the class and nips the problem in the bud. So accidentally deploying the experimental definition with the stable declaration won't pass Jeep's scrutiny either, if changes are at the interface level.

4.3. Variables

Ideally, variables must be implementation details. So they must be protected or private. However, keeping them public is simpler in many cases where classes are not meant to be extended or used by unreliable third parties.

Public variables invariably need read and write access by external functions, and as we all can agree, it is safe to provide interface functions for the task instead of allowing access directly. This usually results in writing a lot of simple boilerplate get and set functions. As with any boilerplate it can be automated away. This is done by using the `get` and `set` directives for the variables. These are the only variable related directives. They generate the getter and setter functions respectively. The generated functions will have the name auto generated based on a simple rule: the first letter of the variable name is capitalized and the word `Get` or `Set` is prefixed as appropriate.

```
let Class = DemoEnv.Object.CreateClass("Class", {
    Variables: {$get_set$__value: 100},
});
```

```
let c = new Class;
cout("c.value: "+c.GetValue())
c.SetValue(-1);
cout("c.value: "+c.GetValue())
```

```
c.value: 100
c.value: -1
```

4.4. Static Members

Static members are the ones that can be accessed without having to instantiate the class. The *Static* property describes the static members of the class. Static functions being not instance specific cannot be virtual or abstract. They also cannot use any instance specific directives, and instance specific flags don't apply to them. The variables cannot use get and set directives either. Static members and public members can have the same names.

```
let Class = DemoEnv.Object.CreateClass("Class", {
    CONSTRUCTOR: function(v){
        this.value = v;
        cout(this.$name+" instantiated with "+this.value);
    },
    Variables: {value: 100},
    Functions: {
        Print: function(){
            cout("The public value given was " + this.value);
            this.$def.Print();
        }
    },
    Static: {
        Print: function(){this.RealPrint()},
        RealPrint: function(){cout("The static value is " + this.value);},
        value: -273
    }
});

let c = new Class(100);
c.Print();
Class.Print();
```

Accessing static information is pretty straight forward. The member functions use the reflective property *$def* and the static functions use the *this* object. Inside static functions, the *this* is effectively the *$def*. This usage of *this* inside static functions is partly the consequence of not relying on global definition objects. It could have been made such that static functions also used *this.$def* to keeps things consistent, but it seemed too much work for a feature that is used only sparingly. Its me being lazy. It takes one to know one.

4.5. Protected Members

Protected members are a bit odd but quite useful. The name is taken from C++ but the semantics and behavior are very different. Protected members offer a simple protection mechanism against accidental usage of members that are meant to be implementation details. Without this, the only way to impose the restriction is with comments to that effect and relying on the tacit programmer's contract.

Consider this example.

```
let Class = DemoEnv.Object.CreateClass("Class", {
    CONSTRUCTOR: function(v){
        this.value = v;
        cout(this.$name+" instantiated with "+this.value);
    },
    Variables: {value: 0},
    Functions: {
        Print: function(){this.$.PrintDetails();},
    },
    Protected: {
        count: -1,
        PrintDetails: function(){
            cout("value (public): " + this.value);
            cout("count (protected): " + this.$.count);
        }
    }
});

let c = new Class(100);
c.Print();
try{c.count=0}catch(e){cout(e.message)}
try{c.PrintDetails()}catch(e){cout(e.message)}
```

```
Class instantiated with 100
value (public): 100
count (protected): -1

JEEP aborting due to run time error. The variable 'count' of the class [Class] is
protected and not accessible directly.
JEEP aborted.

JEEP aborting due to run time error. The function 'PrintDetails' of the class
[Class] is protected and not accessible directly.
JEEP aborted.
```

The protected members are present in the dollar object. If you want, you can interpret the dollar as having to pay in terms of overhead for the added protection, because you do. It will be briefed in ***Internals*** chapter. The protected status of the members can be revoked during generation, though code still uses the dollar object. It somewhat reduces the overhead, but it still costs slightly more than public access.

Note that both public and protected members are accessed with `this` and `$` respectively in both public and protected functions. This is a design choice made to keep things consistent and confusion free. There were at least two other design possibilities 1. using `this` to mean public in public and protected in protected, and using `this.pub` and `this.prot` to disambiguate 2. using `this.prot` and `this.pub` consistently throughout. As you can see, such usages are quite cumbersome and confusing (but easier to implement). The existing design makes you think less about the semantics and focus on the class structure. The choice of using dollar rather than words was made to reduce typing words like `$prot` etc (though it would need using the Shift key).

The exception raised upon attempt to access protected members on the instance is also a design choice. It would have been simpler to not have them. Protected members are not present in the instance. So attempting to use the function would make the engine raise an exception if Jeep hadn't. As for assigning value to the variable, it would mutilate the object and probably fail silently. It would eventually be noticed by the developer and debugged

accordingly, but because Jeep tries to be a robust framework, it saves the hassle in the first place by failing immediately.

Protected members are odd for a few reasons. They exist in a separate object but share the public namespace. This means you cannot have same names for public and protected members. The reason for this is optional revocation of the protected status. Whet it is done, the protected members are added to the instance which is aliased in the dollar object. Public variables being truly implementation details cannot use get and set directives. In contrast, protected functions comply with every syntax rule and have the exact behavior as public functions. They can even be virtual.

In fact, the real motivation for protected members was to safeguard virtual functions. As with other functions, virtual functions can be either interface or implementation. There is no fail proof way to guarantee that the implementation ones are not called out of turn. The consequence of doing that can be quite nasty because virtual functions invariably operate in a controlled context and make many assumptions about it. To remedy this, protected members were introduced.

Note that the dollar object is itself accessible publicly, so external code can access protected members via that. However, the intention is to avoid accidental access, not deliberate access. There is a difference between deliberately accessing implementation details and accidentally accessing them. No one can stop deliberate access in plain JavaScript. Accidental access is a consequence of development tools fueling programmer laziness. Almost every editor worth considering comes with an auto complete feature. If the public and implementation functions start with similar letters, a lazy programmer could accidentally choose the wrong one. This would be noticed eventually and debugged as well, but allowing that is against Jeep's principle. There is a way to solve this problem without Jeep's assistance or relying on programmers' alertness, but that would mean designing the function names keeping these things in mind. We all know ho seriously people take designing APIs. It would simply move the problem from a lazy programmer who codes to a lazy programmer who designs.

4.6. Private Members

The concept of private members (though an unfortunate name) is taken from C+ as is. The kind of privacy created by Jeep is well beyond what is done with plain JavaScript code - making closures with local functions and variables - and offers some important benefits. The private members are accessible by all class member functions, unlike closure method where they are localized. Any attempt to allow global access in plain JavaScript code renders the closure useless. The Jeep method improves performance because, unlike the closures that are created every time the function runs, private members are created exactly once.

Private members look and feel very much like protected members, except that they are inside the double dollar object. If you want, you can think of it as paying more to avail the protection mechanism. You indeed pay more than public access as will be briefed in ***Internals***.

```
let Class = DemoEnv.Object.CreateClass("Class", {
    CONSTRUCTOR: function(v){
        this.value = v;
        cout(this.$name+" instantiated with "+this.value);
    },
    Variables: {value: 0},
    Functions: {
        $usepriv$__Print: function(){
            this.$$.Print();
        },
    },
```

```
    Private: {
        value: -1,
        Print: function(){
            cout("The public value given was " + this.value);
            cout("The private value set was " + this.$$.value);
        }
    }
});

let c = new Class(100);
c.Print();
```

```
Class instantiated with 100
The public value given was 100
The private value set was -1
```

There are some important difference from protected members. Private members exist in a separate object as well as separate namespace. So the names can clash with public and protected members without causing errors. Private functions cannot be virtual. Private member access must be indicated via directives. Only functions that use the `usepriv` directive will get the double dollar object in their `this`. When all functions need to access, it is cumbersome to use the directive everywhere. The flag *`using-private-members`* can be used instead. Constructors cannot use directives, so there is a separate flag `constructor-using-private-members` to allow access from constructors.

```
Class = DemoEnv.Object.CreateClass("Class", {
    Flags: "using-private-members, constructor-using-private-members",
    CONSTRUCTOR: function(v){
        this.value = v;
        this.$$.value = v*3;
        cout(this.$name+" instantiated with "+this.value);
    },
    Variables: {value: 0},
    Functions: {
        Print: function(){
            this.$$.Print();
        },
    },
    Private: {
        value: -1,
        Print: function(){
            cout("The public value given was " + this.value);
            cout("The private value set was " + this.$$.value);
        }
    }
});

c = new Class(100);
c.Print();
```

```
Class instantiated with 100
The public value given was 100
The private value set was 300
```

It is designed that the access must be explicitly indicated because setting up private members has a cost, as mentioned. It seems better to allow the programmer to decide how much to pay rather than charging for all functions when only some use the access.

There is one consequence of the way private members are designed that goes against Jeep's own principle. Because the names can clash with public members, accidentally attempting to access private members would result in accessing public members if names clash. If names don't clash, setting variable values would end up mutilating the object and function access will result in the engine throwing an exception. If these must be avoided, then private members must share the namespace with public and protected members. I feel this is an unfair imposition on the programmer because private members are implementation details and he must be free to use whatever names he wants, within the rules. So, in designing, I chose programmer convenience from design point of view than programmer convenience from laziness point of view. The mnemonic that double dollar means more cost works on multiple levels now.

4.7. Robustness

Robustness is added to classes by enforcing syntactic and semantic restrictions on their functions. In effect, robustness makes the rather weak JavaScript functions strong. Robustness features offer semantic and behavioral guarantees, and assist in debugging by pinpointing to the offender in almost all cases. When not possible to pinpoint, attempt is made to provide as much contextual information as possible.

Suppose an instance suffers from spurious state change. In plain JavaScript code, you would spend a lot of time placing breakpoints and all that. In contrast, when robustness features are enforced, if any code attempts such changes that violates the enabled features, it will be immediately aborted and reported.

All features are directly adapted from C++. Since robustness features are part of the C++ language, violating them causes compile time error. For JavaScript these features are alien, so it cannot be determined till runtime if violation has occurred. Well, it can be done at generation stage by processing the function code, and it is done in an extremely limited way to determine if functions are empty in the split method. But for robustness, it needs much more complex analysis, and I don't want to turn Jeep into a JavaScript source compiler. So violation is considered runtime error and the function is aborted.

Robustness is added via flags and function directives. All features are additive, so you can use combinations to get precise guarantees. They are equally applicable to public protected, private and static functions, except that static functions cannot have instance specific features.

4.7.1. Constant Functions

A constant function is one that cannot modify the member variables. The directive `const` renders a function constant. Since it is instance specific, static functions cannot be constant.

```
Class = DemoEnv.Object.CreateClass("Class", {
    Functions:{
        $const$__Print: function(){cout("Class data: " +this.str)},
        $const$__Test: function(){this.str = "changed"}
    },
    Variables: {str: "original"},
});

c = new Class();
c.Print();
try{c.Test()}catch(e){cout(e.message)}
c.Print();
```

```
Class data: original
JEEP run time error [class Class]. The function 'Test' is declared constant but
tried to modify the variable 'str'.
JEEP aborted.
Class data: original
```

There is a more subtle way of changing the variables than shown in the above example. You could change the first letter of the string for instance. When such deep change occurs to variables, Jeep doesn't trap them to abort; rather it ignores the changes such that the object is left in its original state. The reason for this is simply that trapping such deep changes is prohibitively expensive, even for development environment. Notice the absence of *try catch* block in the following example.

```
Class = DemoEnv.Object.CreateClass("Class", {
    Functions:{
        Print: function(when){cout(when +
               " str: "+this.str+" obj.value: "+this.obj.value)},
        $const$__ChangeString: function(){this.str[0] = "changed"},
        $const$__ChangeObject: function(){this.obj.value = 0}
    },
    Variables: {
        str: "original",
        obj: {value: 100},
    },
});

c = new Class();
c.Print("Before change");
c. ChangeString()
c. ChangeObject()
c.Print("After change");
```

```
Before change str: original obj.value: 100
After change str: original obj.value: 100
```

The constantness restriction is applicable to all the functions in the call chain initiated by the constant function, which includes both member and non member functions. None of the functions in the call chain are allowed to change the objects variables.

Upon violation, the error is always assigned to the initiating constant function simply because there is no way to know which function is being executed at a given time. This is one scenario where the offender cannot be pinpointed.

```
Class = DemoEnv.Object.CreateClass("Class", {
    Functions:{
        $const$__Process: function(callback){
            if(callback)
                callback(this)
            else
                this.process()
        },
        process: function(){this.str = "changed"}
    },
    Variables: {str: "original"},
```

```
});

c = new Class();
try{c.Process()}catch(e){cout(e.message)}
try{c.Process(function(obj){obj.str = "??"})}catch(e){cout(e.message)}
```

```
JEEP run time error [class Class]. The function 'Process' is declared constant but
tried to modify the variable 'str'.
JEEP aborted.

JEEP run time error [class Class]. The function 'Process' is declared constant but
tried to modify the variable 'str'.
JEEP aborted.
```

4.7.2. Internal Variable Change

As the name suggests, this feature restricts the member variable change to happen inside non constant member functions (constant functions remain constant). This is a class definition related feature, so needs a flag rather then directives. The flag *`internal-variable-change`* adds this feature. As with constantness, deep changes are ignored rather than trapped.

```
Class = DemoEnv.Object.CreateClass("Class", {
    Flags: "internal-variable-change",
    Functions:{
        Print: function(when){cout(when+" "+"str: "+this.str)},
        Process: function(){
            this.process();
            cout("Class processed data: " +this.str)
        },
        process: function(){this.str = "changed internally"}
    },
    Variables: {str: "original"},
});

c = new Class();
c.Process()
c.Print("Before external change")
c.str[0] = '?'
c.Print("After external change")
try{c.str = "??"}catch(e){cout(e.message)}
```

```
Class processed data: changed internally
Before external change str: changed internally
After external change str: changed internally
JEEP run time error [class Class]. The variable 'str' was attempted to be modified
externally.
JEEP aborted.
```

This is another scenario where the offender cannot be pinpointed because violations like this happen outside Jeep's purview.

As with constant functions, this restriction applies to all functions in the call chain initiated by member functions. However, in this case it is a privilege rather than a restriction, and it a disadvantage to allow this privilege to external functions. To remedy this, classes can use *`ExternalCall`* function. This function is auto generated for every class that use the access

flag. This function takes two arguments, the function to call and the arguments to pass to that function. This call makes it explicit to both the framework and the reader that the function is invoked without the privilege.

To reiterate, the nuance here is that any external function invoked directly will be able to change the object variables. Such external functions that get this privilege are called *friend functions* (again, a C++ term). Note that the distinction of being a friend is applicable only when this flag is present.

```
Class = DemoEnv.Object.CreateClass("Class", {
    Flags: "internal-variable-change",
    Functions:{
        Process: function(callback, friend){
            if(callback)
            {
                if(friend) callback("friend-call")
                else this.ExternalCall(callback, "external-call")
            }
            else
                this.process();
            cout("Class processed data: " +this.str)
        },
        process: function(){this.str = "changed internally"}
    },
    Variables: {str: "original"},
});
c = new Class();
let proc = function(p){c.str = "changed "+p}
c.Process()
c.Process(proc, true)
try{c.Process(proc, false)}catch(e){cout(e.message)}
```

```
Class processed data: changed internally
Class processed data: changed friend-call
JEEP run time error [class Class]. The variable 'str' was attempted to be modified
externally. JEEP aborted.
```

4.7.3. Function Argument Constantness

This feature renders the arguments passed to the functions constant. The `argconst` directive enforces this on the function. Any attempt to change the arguments is not trapped, rather the changes are ignored. The reason is two fold. For deep changes it is again prohibitively expensive, as with constant function and internal variable change features. For direct changes, the JavaScript language comes in the way, in that there is no way you can pass arguments directly and access arguments directly by name while still having constantness enforced in both development and production modes.

```
Class = DemoEnv.Object.CreateClass("Class", {
    Functions:{
        $argconst$__Test: function(value, obj){
            value = -99;
            obj.value = -99;
            cout("Inside function- value: "+value+" obj: "+JSON.stringify(obj));
        }
    },
});

c = new Class();
let value = 0, obj = {value: 0};
```

```
cout("Before- value: "+value+" obj: "+JSON.stringify(obj));
c.Test(value, obj)
cout("After- value: "+value+" obj: "+JSON.stringify(obj));
```

```
Before- value: 0 obj: {"value":0}
Inside function- value: -99 obj: {"value":-99}
After- value: 0 obj: {"value":0}
```

4.7.4. Function Argument Count

The directive *argnum* add this feature which enforces that a function be invoked with exactly the number of arguments as present in the class description.

This feature goes against the grain of JavaScript but it is necessary to write solid reliable software. Moreover, many a code add ad hoc tests for argument count, so this feature is not as alien as it might seem.

```
let Class = DemoEnv.Object.CreateClass("Class", {
    Functions:{
        $argnum$__Test: function(a, b){cout("args: " +a+" "+b)}
    },
});

let c = new Class();
c.Test(1, 2)
try{c.Test()}catch(e){cout(e.message)}
try{c.Test(1)}catch(e){cout(e.message)}
try{c.Test(1,2,3)}catch(e){cout(e.message)}
```

```
args: 1 2
JEEP run time error [class Class]. The function 'Test' is declared to take 2
arguments but invoked with 0.
JEEP aborted.
JEEP run time error [class Class]. The function 'Test' is declared to take 2
arguments but invoked with 1.
JEEP aborted.
JEEP run time error [class Class]. The function 'Test' is declared to take 2
arguments but invoked with 3. JEEP aborted.
```

4.7.5. Function Argument Type

This is another feature going against the grain of JavaScript but is necessary to write solid reliable software. It enforces types on arguments. It also helps self documenting the code.

The argument type information would be quite cumbersome, and sometimes impossible, if used as directives. Hence it is moved to a string, which has the name same as the function it is linked to but with a dollar character prefixed. This string is called *function argument type descriptor*. It must appear in the functions descriptor only. A special exception is made to accommodate this inside functions. It contains comma separated names of the types expected.

This feature is quite elaborate so I will start with a simple example.

```
Class = DemoEnv.Object.CreateClass("Class", {
    Functions:{
        $Print: "String, Number",
        Print: function(a,b){cout("args: "+a+" "+b)}
```

```
    },
});

c = new Class();
c.Print("twice", 2)
try{c.Print()}catch(e){cout(e.message)}
try{c.Print(1)}catch(e){cout(e.message)}
try{c.Print("a", 3, 4)}catch(e){cout(e.message)}
```

```
args: twice 2

JEEP run time error [class Class]. The function 'Print' expects 2 number of
argument(s).
JEEP aborted.

JEEP run time error [class Class]. The function 'Print' expects 'string' type for
argument 0.
JEEP aborted.

args: a 3
```

When no arguments are given, the feature acts as if it were an argument count validation. When there are arguments, each one is validated. If more arguments than expected are given, this feature does nothing because the count validation is not its job; what it does with zero arguments is inevitable. In the split method, the same descriptor must appear in both places or there must be none in both places; anything else is an error. The descriptor can appear anywhere within the function description, but it is good to have it just above the function declaration (and definition). Doing this makes the code more readable and meaningful.

There are four basic types that correspond to JavaScript native objects – Number, String, Array and Object. Note that the first letters are capitalized. The validation is only superficial for arrays and objects - you cannot specify array of numbers etc, or the object layout. The reason is simply the prohibitive cost of such validation. As with some other validations, this is only a first line of defense; any deeper test needed must be done by the function.

There is another type represented by a single question mark. It allows any type. This is useful for functions where one parameter suggests the type and another is data of that type.

Then, as you might expect, the type system extends to Jeep objects as well. You can even directly mention if the type is a record, structure or a class.

```
// the class described
Class = DemoEnv.Object.CreateClass("Class", {
    Functions: {
        $Print: "Array, Number, Object, String,
                 record.Record, struct.Struct, class.MyLib.Class, ?",
        Print: function(arr, num, obj, str, rec, st, cl, x){
            cout("Class.Print")
            st.Print();
            cls.Print();
            cout("x: "+typeof x)
        },
    }
})
```

```
// the types created elsewhere
function print(){cout(this.$name+".Print")}

DemoEnv.Object.RegisterRecord("Record", {dummy: 0})
DemoEnv.Object.RegisterStruct("Struct", {
    Functions: {Print: print},
    Variables: {dummy: 0}
})

let Lib = DemoEnv.CreateNamespace("MyLib")
Lib.RegisterClass("Class", {Functions: {Print: print}})

let Record = JEEP.GetRecord("Record");
let Struct = JEEP.GetStruct("Struct");
let LibClass = Lib.GetClass("Class");
```

```
// the class being used
c = new Class;
let r = new Record.New()
let s = Struct.New()
let cls = new LibClass;

c.Print([1],2,{},"",r,s,cls,10)
c.Print([1],2,{},"",r,s,cls,"")
c.Print([1],2,{},"",r,s,cls,[])
c.Print([1],2,{},"",r,s,cls,{})
```

```
Class.Print
Struct.Print
MyLib.Class.Print
x: number
...
x: string
...
x: object
...
x: object
```

For jeep objects, the type must be prefixed with the object type name. This is a simple and effective mechanism to mention the exact types. Note that objects in namespace must be mentioned with fully qualified names. There is just no simpler way to do both of this than typing it all out.

The jeep object validation is a two step process. The object is first accessed and then the *`InstanceOf`* is executed. This implies that only registered objects can be used in typing. It is this way because there is no way for the framework to access a locally created object to validate the type. This is another advantage of registering objects rather then creating them.

Consequently, mentioning unregistered Jeep objects in the type descriptor is an error.

```
Class = DemoEnv.Object.CreateClass("Class", {
    Functions:{
        $Print: "record.Record, class.Number",
        Print: function(a,b){cout("args: "+a+" "+b)}
    },
});

c = new Class();
```

```
try{c.Print(1,2)}catch(e){cout(e.message)}
try{c.Print(r,2)}catch(e){cout(e.message)}
```

```
JEEP run time error [class Class]. The function 'Print' expects 'record Record'
type for argument 0.
JEEP aborted.

JEEP run time error [class Class]. The function 'Print' expects 'class Number' type
for argument 1 but it is not registered.
JEEP aborted.
```

4.8. Constructor

A class constructor has the basic behavior same as a structure constructor. However, due to added complexity of classes, a class constructor has more features.

4.8.1. Copy Constructor

A copy constructor is a constructor takes one argument and copies it to the instance, effectively cloning it. At least that is what is applicable to C++. In Jeep, the results are same but the mechanism is different.

Unlike C++ where the programmer must define a copy constructor, Jeep implements the mechanism automatically without the programmer having to write extra code. There is no need to write a constructor taking one argument and using `InstanceOf` on it before proceeding to clone it. In fact, when copy construction happens, the constructor will not be called because it is not necessary. Copy construction happens when there is exactly one argument given during instantiation and it happens to be an instance of the class; otherwise the arguments are passed to the constructor as usual.

```
DemoEnv.Object.RegisterClass("CopyConstructorDemo", {
    CONSTRUCTOR: function(s){
        this.str = s;
        cout("MyClass.CONSTRUCTOR "+this.str)
    },
    Functions:{
        Append: function(s){this.str += "+"+s},
        Print: function(){cout(this.str)}
    },
    Variables: {str: ""},
});

let Class = JEEP.GetClass("CopyConstructorDemo");
let c = new Class("instance");
let d = new Class(c);
d.Append("copied");
c.Print();
d.Print();
let e = new Class(c+"copied");// not copy constructor
let f = new Class(c, d);// not copy constructor
```

```
MyClass.CONSTRUCTOR instance
instance
instance+copied
MyClass.CONSTRUCTOR [object Object]copied
MyClass.CONSTRUCTOR [object Object]
```

4.8.2. Constructor Failure

The instantiation fails if the constructor fails. A constructor fails when exceptions are thrown. The exception could be thrown by the engine due to bad code, or it could be a custom exception thrown by the class. Custom exceptions can be replaced with a simple returning of `false` to indicate failure in a controlled way. Doing this will also have performance benefits.

```
DemoEnv.Object.RegisterClass("CtorFailureDemo_Strike", {
    CONSTRUCTOR: function(s){
        throw new Error("Strike!!")
        this.str = s;
        cout("CtorFailureDemo_Strike.CONSTRUCTOR "+this.str)
    },
});

DemoEnv.Object.RegisterClass("CtorFailureDemo_BadSyntax", {
    CONSTRUCTOR: function(s){
        s.Change(0);
        this.str = s;
        cout("CtorFailureDemo_BadSyntax.CONSTRUCTOR "+this.str)
    },

});
DemoEnv.Object.RegisterClass("CtorFailureDemo_FalseReturn", {
    CONSTRUCTOR: function(s){
        this.str = s;
        cout("CtorFailureDemo_FalseReturn.CONSTRUCTOR "+this.str);
        return false
    },
});

let Class = JEEP.GetClass("CtorFailureDemo_Strike");
try{new Class("Hello")}catch(e){cout(e.message)}

Class = JEEP.GetClass("CtorFailureDemo_BadSyntax");
try{new Class("Hello")}catch(e){cout(e.message)}

Class = JEEP.GetClass("CtorFailureDemo_FalseReturn");
try{new Class("Hello")}catch(e){cout(e.message)}
```

```
The class 'CtorFailureDemo_Strike' failed to instantiate due to the exception
<Error: Strike!!>.

The class 'CtorFailureDemo_BadSyntax' failed to instantiate due to the exception
<TypeError: s.Change is not a function>.

CtorFailureDemo_FalseReturn.CONSTRUCTOR Hello
The class 'CtorFailureDemo_FalseReturn' failed to instantiate.
```

When failure is due to a returning false, nothing extra is mentioned in the message. Note that instantiation fails when any constructor in the hierarchy fails in any form. More about this is explained while explaining hierarchy in the next chapter.

4.11. Destructor and Instance Lifetime

Destructor is the counterpart to constructor. It is the last function called before the object is deleted. While central to C++, the concept of destructor is alien to JavaScript like languages that use automatic garbage collection mechanism. In this mechanism, destruction happens when nobody cares about the object anymore (no doubt objects are weak in JavaScript), but this happens in the engine code and there is no way for the script code to know of the destruction happening. There is no place to say the final goodbye or f*** you.

The lack of destructors has two implications. First, there is no guarantee as to when the instance is destroyed. Second, and related, there is no place to do the last piece of action like releasing the resources. The first reason is more to do with performance guarantees and is not relevant to JavaScript. It is the second one that is interesting. If the resource is a string or some other native data type, it is not an issue. However, what if it is a database connection? I have no idea how it is done in JavaScript (probably with a lot of effort), but in C++ it is the destructor that comes to the rescue. A destructor also allows us to be lazy, in that we can put the cleanup code in it and rest assured that it will be called no matter what. Jeep brings this power to JavaScript, in a limited way.

In C++, the destruction happens when an instantiated object goes out of scope (I am deliberately ignoring pointer scenario). The scope rule being a language feature in C++, the destructor invocation code is generated automatically. Though such a rule exists in JavaScript, it applies to the engine and not the script. To enjoy the benefits of destructors in the script, we have to create an explicit scope inside which we instantiate the classes such that the destruction is overridden by our code.

This example makes more sense with two classes but I am using the same class to reduce code. There are many new details shown, more than usual, so consider it for a while.

```
let Class = DemoEnv.Object.CreateClass("Class", {
    CONSTRUCTOR: function(s){
        this.str = s;
        cout("Class.CONSTRUCTOR "+this.str)
    },
    DESTRUCTOR: function(){
        cout("Class.DESTRUCTOR "+this.str)
    },
    Functions:{
        Append: function(s){this.str += "+"+s},
        $scoped$__Test: function(){
            this.$def.ScopedCreate("destructor-enabled")
        },
        $scoped$__TestFail: function(){
            this.$def.ScopedCreate("destructor-enabled")
            this.Run();
        }
    },
    Variables: {str: ""},
});

let c = new Class("destructor-not-enabled");
c.Test();
try{c.TestFail()}catch(e){cout(e.message)}
```

```
Class.CONSTRUCTOR destructor-not-enabled
Class.CONSTRUCTOR destructor-enabled
Class.DESTRUCTOR destructor-enabled
Class.CONSTRUCTOR destructor-enabled
```

```
Class.DESTRUCTOR destructor-enabled
this.Run is not a function
```

Firstly, it is important to realize that destruction mechanism is an instance specific behavior and not class specific. While a class that wants its instances to have destructor mechanism must be described in a certain way, availing the benefit depends on how you instantiate.

The mechanism is enabled only when instantiated via the *`ScopedCreate`* function. While the *`new`* operator could have been reused similar to init constructor, it seemed better to have an explicitly different function for the purpose. This function is static and automatically generated for all classes that define a destructor. This function must be used in specific locations only; using elsewhere results in errors. To be used inside member functions, the functions must have the *`scoped`* directive. The best part and the USP of the destruction mechanism is that it is invoked no matter what. Notice that the destructor was invoked even when *`TestFail`* caused exception due to bad code.

If you are a native JavaScript developer, or are struggling to see the utility of this mechanism, in JavaScript, consider this representative example.

```
let Logger = DemoEnv.Object.CreateClass("Logger", {
    DESTRUCTOR: function(){
        cout("Logging the following: "+this.logs.join())
    },
    Functions:{
        Append: function(s){this.logs.push(s)},
    },
    Variables: {logs: []},
});

let App = DemoEnv.Object.CreateClass("App", {
    Functions:{
        $scoped$__Work: function(){
            let logger = Logger.ScopedCreate();
            logger.Append("first")
            logger.Append("second")
            logger.Append("third")
            throw new Error("simulated exception from some function")
        },
    },
});

let a = new App;
try{a.Work()}catch(e){cout(e.message)}
```

```
Logging the following: first,second,third
Error: simulated exception from some function
```

Remember, an exception need not always abort the script. Now imagine guaranteed logging of the information in every situation with normal JavaScript. The code would be replete with exception handling and other defensive coding, The destructor mechanism not only improves the readability, and the performance, but also guarantees correct behavior in all scenarios.

Now, what might happen if the destructor throws an exception when employed in scoped functions? In C++ it is an unrecoverable situation because destructor itself is supposed to be the failsafe, so the program aborts. But in Jeep the exception is ignored. The only thing that happens is the destructor failing to do the task it was intended to do.

```
let D = DemoEnv.Object.CreateClass("DtorThrow", {
    CONSTRUCTOR: function(){cout("DtorThrow instantiated")},
    DESTRUCTOR: function(){
        throw new Error("destructor exception")
    },
    Functions:{
        $scoped$__Work: function(){
            this.$def.ScopedCreate();
            throw new Error("simulated exception from some function")
        },
    },
});
let d = new D;
try{d.Work()}catch(e){cout(e.message)}
cout("other code")
```

```
DtorThrow instantiated
DtorThrow instantiated
simulated exception from some function
other code
```

5. Functions API

Many of the class member function related features are applicable beyond the class setup. The exceptions are instance specific features like constant functions. These features can be used to augment any plain JavaScript function. The API present in *`Function`* object of the environment does exactly that. It is assumed that you have read the class chapter and understand what the features are about, so I will just demonstrate the API.

5.1. Scoping

The scoping mechanism mentioned in class chapter is probably more useful for non member functions, so there are two versions of the API to achieve the exact behavior.

```
let Class = DemoEnv.Object.CreateClass("Class", {
    CONSTRUCTOR: function(s){
        this.str = s;
        cout("Class.CONSTRUCTOR "+this.str)
    },
    DESTRUCTOR: function(){
        cout("Class.DESTRUCTOR "+this.str)
    },
    Variables: {str: ""},
});

let func = DemoEnv.Function.MakeScoped(function(s){
    Class.ScopedCreate(s)
    return 10;
});

let k = func("destructor-enabled-via-MakeScoped");
    let j = DemoEnv.Function.ScopedCall(function(s){
    Class.ScopedCreate(s)
    return 100;
}, "destructor-enabled-via-ScopedCall");

cout("MakeScoped returned: "+k+" ScopedCall returned: "+j)
```

```
Class.CONSTRUCTOR destructor-enabled-via-MakeScoped
Class.DESTRUCTOR destructor-enabled-via-MakeScoped

Class.CONSTRUCTOR destructor-enabled-via-ScopedCall
Class.DESTRUCTOR destructor-enabled-via-ScopedCall

MakeScoped returned: 10 ScopedCall returned: 100
```

Both the functions take a function as their argument. The *`MakeScoped`* function returns another function that has been setup the scoping mechanism. This is useful when the function is to be used multiple times, at different places etc. In contrast, the *`ScopedCall`* function calls the given function immediately with the given arguments. The arguments are given as the second parameter.

This is the only feature that has the immediate call API. This is because it is not a robustness feature. For those features it is absurd to have this because, instead of setting up a function and calling it immediately, you can call it with the correct arguments directly.

5.2. Constant Argument

The function *MakeArgConst* achieves the effect. It takes a function as its argument and returns a function that is setup to treat the arguments as a constant.

```
let obj = {
    a: 100,
    SetA: function(a){this.a = a},
    GetA: function(){return this.a}
}

cout("before call: " + obj.a);

let f = DemoEnv.Function.MakeArgConst(function(obj){
    cout("in call before mod: " + obj.GetA());
    obj.SetA(3);
    cout("in call after mod: " + obj.a);
});

f(obj);

cout("after call: " + obj.a);
```

```
before call: 100
in call before mod: 100
in call after mod: 3
after call: 100
```

5.3. Argument Count Validation

The function *MakeArgNumValidated* achieves the effect. It takes a function as its argument and returns a function that is setup to validate the argument count.

```
func = DemoEnv.Function.MakeArgNumValidated(function printer(what, where){
    cout(what + " in " + where);
})

func("puss", "boots")

try{func()}catch(e){}
```

```
puss in boots
JEEP run time error [class <unknown>]. The function 'printer' is declared to take 2
arguments but invoked with 0.
```

It is enforced that the function have a name so that the error messages issued in case of violation be helpful. There is no use in making such functions and pass them around only to see a <*unknown*> mentioned for the function name.

5.4. Argument Type Validation

The function *MakeArgTypeValidated* achieves the effect. It takes two arguments, the type descriptor and the function itself, and returns a function that is setup to validate the argument type.

```
func = DemoEnv.Function.MakeArgTypeValidated(
    "String, Number",
    function printer(a,b){
        cout("args: "+a+" "+b)
    });

func("abc", 10)

try{func(10, 20)}
catch(e){}
```

```
args: abc 10
JEEP run time error [class <unknown>]. The function 'printer' expects 'string' type
for argument 0.
```

It is enforced that the function have a name so that the error messages issued in case of violation be helpful. There is no use in making such functions and pass them around only to see a <*unknown*> mentioned for the function name.

6. Single Inheritance

This chapter is about class hierarchies. Only classes can be part of a hierarchy. As you already know, Jeep supports both single and multiple inheritance. Conceptually, multiple inheritance is a simple extension to single inheritance, but with some not so simple additional constraints and consequences. So I will first begin with single inheritance in this chapter and then build on it to discuss multiple inheritance in the next.

6.1. The Basics

To inherit, a class *derives* from another class. The class that derives is called the *derived class*, and the class that is derived from is called the *base class*. The class that starts the hierarchy is called the *root class*. A derived class inherits all public and protected members from its base class. Private members of a class remain private and inaccessible to derived classes. All inherited members are accessible as if it were the derived class' own. I often use the term *extend* with respect to derived class to mean *derive and extend by adding additional stuff*. Thereby, I give *derive* a structural meaning and *extend* a functional meaning. Further, I make a distinction between *base class* and *ancestor class*, the former being the one mentioned in the class description and the latter being any base up in the hierarchy.

Every class in the hierarchy, except the root class, has a reflective property `$base` through which the class can access the base class functions. This property contains sub objects bearing the same name as the base classes. Inside these objects are found the base class functions that are overridden by the derived class. Unlike C++, overriding is restricted to virtual functions in Jeep. Overriding happens when a derived class reimplements a base class virtual function. The property can also contain base class constructor based on flags. The creation of this property and the objects inside it are controlled with directives and flags.

It is called single inheritance when a class has only one base. That base could have its own base and so forth, but as long as every base has only one base, it remains single inheritance. Deriving is simple – just add the base class name to the `BaseClass` property of the class description object which is a comma separated string.

```
DemoEnv.Object.RegisterClass("Base", {
    CONSTRUCTOR: function(){cout("Base.CONSTRUCTOR")},
    Functions: {
        Work: function(){cout("working...")}
    }
})

DemoEnv.Object.RegisterClass("Derived", {
    BaseClass: "Base",
    CONSTRUCTOR: function(){cout("Derived.CONSTRUCTOR")},
})

let Derived = JEEP.GetClass("Derived")
let d = new Derived;
d.Work();
```

```
Base.CONSTRUCTOR
Derived.CONSTRUCTOR
working...
```

The base class constructor is always called before the derived class constructor. The process of construction begins from the root class and ends at the leaf class, which is the derived class. Inheritance merges all classes into one. The instance can be thought of as a vertical arrangement of Lego bricks, with each brick being a base. If the one at the bottom is the root class and the one at the top is the derived class, construction builds the tower bottom up.

One consequence of the merging is that reflective properties will always be that of the derived class no matter which part of the instance accesses it. That is why the base class explicitly mentions its name in the message and not use `$name`. This is unlike C++ where classes are not merged and every base retains its own set of variables. The difference is due to Jeep having to work with whatever is possible in JavaScript.

Using the base class name in the `BaseClass` property implies that the base classes must be registered. While this is beneficial most of the time because library classes are what are usually derived from, there is still a need to create a hierarchy that is local to a scope and not needed by others. This is similar to why `CreateX` would be used. To build such hierarchies, you have to use another class descriptor `CrBaseClass`. This is an array of the generated class definitions. Note that by combining both the base class descriptors, you can create local extensions of library classes, but that constitutes multiple inheritance.

```
Base = DemoEnv.Object.CreateClass("Base", {
    CONSTRUCTOR: function(a){
        cout("Base.Constructor args: "+a)
    },
});

Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [Base],
    CONSTRUCTOR: function(a){
        cout("Derived.Constructor args: "+a)
    },
});

new Derived(10);
```

```
Base.Constructor args: 10
Derived.Constructor args: 10
```

Single inheritance imposes one constraint - public and protected member names cannot repeat in a hierarchy. This implies that private names can. So, if a class has a variable called *value*, its derived class cannot have a variable of the same name. This is necessary to remove conflict. Note that C++ doesn't have this constraint, and there is a way to disambiguate which variable we are referring to. However, according to me, that is a bad solution to the problem, and creates more ambiguity than it tries to solve. Banning it is a simpler and more effective solution. Further, as inheritance is an upward growing structure, with new code added above old code, new code can simply avoid using conflicting names.

6.2. Constructor and Destructor

The base constructor call is automatically made by the framework but it can be overridden with flags. This will be explained at the end of this section because it needs another mechanism to be understood first. When copy construction happens, none of the constructors are called. This behavior is similar to what happens with a solitary class. When init

construction happens, the constructor calls are made, as is done with solitary classes. Due to the merging of classes, the init object can now have all the variables from the ancestor classes.

```
Base = DemoEnv.Object.CreateClass("Base", {
    CONSTRUCTOR: function(a){
        cout("Base.Constructor args: "+a)
        cout("Base.Constructor name: "+this.GetName()+" age: "+this.GetAge())
    },
    Variables: {
        $get$__name: "?",
        $get$__age: 0,
    }
});

Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [Base],
    CONSTRUCTOR: function(a){
        cout("Derived.Constructor args: "+a)
        cout("Derived.Constructor city: "+this.GetCity())
    },
    Variables: {
        $get$__city: "?",
    }
});

new Derived("init", {name: "unknown", age: 100, city: "unknown"})
```

```
Base.Constructor args: undefined
Base.Constructor name: unknown age: 100
Derived.Constructor args: undefined
Derived.Constructor city: unknown
```

Destructors when present are called in the opposite order. There are two scenarios where destructors are called. One is the scoped creation as explained with respect to solitary classes. For hierarchies, it just extends to calling the base destructors.

```
Base = DemoEnv.Object.CreateClass("Base", {
    CONSTRUCTOR: function(){cout("Base.CONSTRUCTOR")},
    DESTRUCTOR: function(){cout("Base.DESTRUCTOR")},
});

Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [Base],
    CONSTRUCTOR: function(){cout("Derived.CONSTRUCTOR")},
    DESTRUCTOR: function(){cout("Derived.DESTRUCTOR")},
});

DemoEnv.Function.ScopedCall(function(){
    Derived.ScopedCreate();
})
```

```
Base.CONSTRUCTOR
Derived.CONSTRUCTOR
Derived.DESTRUCTOR
Base.DESTRUCTOR
```

The other scenario when a destructor is called is when construction fails. When any base fails to construct, the instantiation process is abandoned. But Jeep does more to help with resource management. Unlike C++, where such abandonment leads to memory leak, and need convoluted RAII code (a C++ idiom) to cleanup, Jeep employs partial destruction to remedy the problem.

All the bases that successfully constructed will be allowed to destruct. Though the name sounds complex, there is nothing special about partial destruction. A base anyway does not (and should not in well designed classes) know whether it is being constructed and destructed for its own instantiation or for one of its derived class, so partial destruction must not pose a problem. Partial destruction is always attempted, in that Jeep checks if a base class has destructor and, if it does, calls it. So, it is ok if the hierarchy doesn't have destructors.

```
let TopBase = DemoEnv.Object.CreateClass("TopBase", {
        CONSTRUCTOR: function(){cout("TopBase.CONSTRUCTOR")},
        DESTRUCTOR: function(){cout("TopBase.DESTRUCTOR")},
});

let MidBase = DemoEnv.Object.CreateClass("MidBase", {
        CrBaseClass: [TopBase],
        CONSTRUCTOR: function(){cout("MidBase.CONSTRUCTOR")},
        DESTRUCTOR: function(){cout("MidBase.DESTRUCTOR")},
});

let LowBase = DemoEnv.Object.CreateClass("LowBase", {
        CrBaseClass: [MidBase],
        CONSTRUCTOR: function(){cout("LowBase.CONSTRUCTOR failing");return false},
        DESTRUCTOR: function(){cout("LowBase.DESTRUCTOR")},
});

let Derived = DemoEnv.Object.CreateClass("Derived", {
        CrBaseClass: [LowBase],
        CONSTRUCTOR: function(){cout("Derived.CONSTRUCTOR")},
        DESTRUCTOR: function(){cout("Derived.DESTRUCTOR")},
});

try{new Derived;}catch(e){cout(e.message)}
```

```
TopBase.CONSTRUCTOR
MidBase.CONSTRUCTOR
LowBase.CONSTRUCTOR failing
MidBase.DESTRUCTOR
TopBase.DESTRUCTOR
The class [Derived] failed to instantiate at base [LowBase].
```

Partial destruction might look odd but it is very useful when classes manage resources. Suppose a base class holds a connection to a web service that it will release during destruction because the class and its descendents were meant to be created inside a scope. The expected scope behavior won't be realized in case of construction failure because the instantiation never happened. So, the resource held by a higher base will leak due to a lower base failing. Of course, in the script the GC would clear the base class variable and the web service probably times out. So there would be no leak. However these things happen eventually, at some time in the future. But it is good for actions to happen when intended. Partial destruction helps achieving the ideal. It augments the already powerful destruction mechanism.

As mentioned, the default order of construction can be overridden. A class can instruct the framework that it will invoke the base class constructors manually by using the flag *manual-base-construction*. I have to admit that I myself am struggling to see the utility of this, but you never know what kind of out of order construction people want for their idiosyncratic design. So this exists for the sake of flexibility only. However, using this flag is playing with fire and not recommended.

Consider this scenario.

```
let TopBase = DemoEnv.Object.CreateClass("TopBase", {
    CONSTRUCTOR: function(){cout("TopBase.CONSTRUCTOR");throw "error";},
    DESTRUCTOR: function(){cout("TopBase.DESTRUCTOR")},
});

let MidBase = DemoEnv.Object.CreateClass("MidBase", {
    CrBaseClass: [TopBase],
    CONSTRUCTOR: function(){cout("MidBase.CONSTRUCTOR")},
    DESTRUCTOR: function(){cout("MidBase.DESTRUCTOR")},
});

let LowBase = DemoEnv.Object.CreateClass("LowBase", {
    CrBaseClass: [MidBase],
    CONSTRUCTOR: function(){cout("LowBase.CONSTRUCTOR failing");return false},
    DESTRUCTOR: function(){cout("LowBase.DESTRUCTOR")},
});

let Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [LowBase],
    CONSTRUCTOR: function(){
        cout("Derived.CONSTRUCTOR");
        this.$base.MidBase.CONSTRUCTOR();
        this.$base.LowBase.CONSTRUCTOR();
        this.$base.TopBase.CONSTRUCTOR();
    },
    DESTRUCTOR: function(){cout("Derived.DESTRUCTOR")},
});

DemoEnv.Function.ScopedCall(function(){
    try{Derived.ScopedCreate();}catch(e){cout(e.message)}
});
```

```
Derived.CONSTRUCTOR
MidBase.CONSTRUCTOR
LowBase.CONSTRUCTOR failing
TopBase.CONSTRUCTOR
The class [Derived] failed to instantiate due to the exception <error>.
```

As you can see from the result, partial destruction doesn't happen. Further, the failure of the *LowBase* goes unnoticed due to the programmer error. If it were not for the *TopBase* failing via exception, the instance would have been constructed wrongly and might have behaved in some very interesting ways. Note that the error message doesn't mention the base that failed unlike the default case.

Then there is the risk of one of the bases itself employing manual construction. This might lead to some bases being constructed more than once and some never at all. A lot of attention to detail is necessary when doing manual construction, and this just explodes with multiple inheritance. As a principle, do not use manual base construction when you have destructors or are deriving from third party classes (even if you have read the implementation details).

6.3. Polymorphism

Polymorphism is the behavior brought about by using virtual and abstract functions. A virtual function is a member function setup such that calling it in the base class invokes the implementation in the derived class if present; else base class function is used. An abstract function is virtual function whose presence obligates derived classes to provide the implementation. A class with unimplemented abstract functions cannot be instantiated. Both these are taken directly from C++, except that abstract functions are called *pure virtual functions* in C++.

There is much more to the definition in C++ that distinguishes polymorphism from something called name hiding, but for JavaScript this is all that is necessary. Further, name hiding and virtual functions coincide in JavaScript. Since JavaScript is simply a big key-value map masquerading as a programming language, the mechanism of virtual functions is sort of inbuilt. But that is only consequential. Jeep offers so much more. It provides structural and semantic guarantees that help with software engineering.

6.3.1. Virtual Functions

```
let TopBase = DemoEnv.Object.CreateClass("TopBase", {
    Functions: {
        TopBaseWork: function(){cout("TopBaseWork details: "
                                              +this.getTopWorkDetails())},
        $virtual$__getTopWorkDetails: function(){return "<from top base>"}
    },
});

let MidBase = DemoEnv.Object.CreateClass("MidBase", {
    CrBaseClass: [TopBase],
    Functions: {
        MidBaseWork: function(){cout("MidBaseWork details: "
                                              +this.getMidWorkDetails())},
        $virtual$__getMidWorkDetails: function(){return "<from mid base>"},
        $virtual$__getTopWorkDetails: function(){return "<from mid base>"}
    },
});

let Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [MidBase],
    Functions: {
        $virtual$__getTopWorkDetails: function(){
            return "<from derived>" + this.$base.MidBase.getTopWorkDetails();
        }
});

let c = new Derived;
c.TopBaseWork();
c.MidBaseWork();
```

```
TopBaseWork details: <from derived><from mid base>
MidBaseWork details: <from mid base>
```

The directive `virtual` renders a function virtual. Once a function is declared virtual, it must always be declared virtual in the hierarchy. This rule exists to help spot virtual functions no matter how deep they appear in the hierarchy. This is unlike C++ where the derived class need not repeat the declaration. A virtual function is central to a class hierarchy and I find it is good to make such functions visible everywhere. Admittedly, this redeclaration won't tell you where the function originated, but it is still better than being completely in the dark.

Further, if a virtual (or abstract) function uses argument type descriptor, it is obligatory for all derived classes to use the type descriptor if they implement the functions. The reason for this is the same as having to use the virtual directive. Note that any class in the hierarchy can decide to start using argument type descriptor for a virtual function, and it need not be a virtual function that the class introduced to the hierarchy. Doing this makes it obligatory for all its descendents only, and not for its bases or siblings.

When a derived class overrides a base class virtual function, the base class function is available to the derived class in the *$base* property inside the appropriate object. This is useful when the overridden function of the derived class must do some preprocessing or post processing that is same as what the base class does. By default, only the immediate base class members are available. If you need remote base functions too, it can be got by using the flag *need-ancestor-access*. I can't see a reason why a well designed class must bypass its immediate base and call one of the remote base's overridden function. However, you never know what people design, so this flag exists just for the sake of completeness.

```
Derived = DemoEnv.Object.CreateClass("Derived", {
    Flags: "need-ancestor-access",
    CrBaseClass: [MidBase],
    Functions: {
        $virtual$__getTopWorkDetails: function(){
            return "<from derived>" +
            this.$base.MidBase.getTopWorkDetails() +
            this.$base.TopBase.getTopWorkDetails();
        }
    }
});

c = new Derived;
c.TopBaseWork();
```

```
TopBaseWork details: <from derived><from mid base><from top base>
```

Retaining the overridden base functions has a tiny penalty in terms of performance and overhead; having ancestor access penalizes slightly more. If a virtual function doesn't need to use the base implementation, the *replace* directive can be used. As the name suggests, this directive causes the derived function to structurally replace the base function such that the base function no longer exists for access. This is a class specific replacement and won't affect a class that derives from such a class.

The relevant base objects in the *$base* property are created only when the derived class needs to access the overridden function, or does manual construction. So if default construction is enabled and all virtual functions are replaced, the base object will not be created. If none of the base class objects need to be created, the *$base* property itself will not be created.

```
let TopBase = DemoEnv.Object.CreateClass("TopBase", {
    Functions: {
        TopBaseWork: function(){cout("TopBaseWork details:\nname: "+
                                this.getName()+"\nid: "+this.getID())},
        $virtual$__getName: function(){return "<name from top base>"},
        $virtual$__getID: function(){return "<id from top base>"}
    },
```

```
});

let Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [TopBase],
    Functions: {
        $virtual$replace$__getID: function(){
            return "<id from derived>" +
                    (this.$base.TopBase.getID ? this.$base.TopBase.getID() : "")
        },
        $virtual$__getName: function(){
            return "<name from derived>" +
                (this.$base.TopBase.getName ? this.$base.TopBase.getName() : "")
        }
    }
});

let c = new Derived;
c.TopBaseWork();
```

```
TopBaseWork details:
name: <name from derived><name from top base>
id: <id from top base>
```

The flag *`replace-virtual-functions`* can be used when all need to be replaced.

```
Derived = DemoEnv.Object.CreateClass("Derived", {
    Flags: "replace-virtual-functions",
    CrBaseClass: [TopBase],
    Functions: {
        $virtual$replace$__getID: function(){
            return "<id from derived>" +
            (this.$base && this.$base.TopBase.getID ?
                        this.$base.TopBase.getID() : "")
        },
        $virtual$__getName: function(){
            return "<name from derived>" +
            (this.$base && this.$base.TopBase.getName ?
                      this.$base.TopBase.getName() : "")
        }
    }
});

c = new Derived;
c.TopBaseWork();
```

```
TopBaseWork details:
name: <name from derived>
id: <id from top base>
```

Replacing virtual functions and the resulting non creation of the *`$base`* property doesn't have any noticeable performance improvement unless you create a lot of instances and access the virtual functions thousands of times per second, or something done at that order. Yet, as a principle, you must always consider replacing wherever it is semantically relevant.

6.3.2. Abstract Functions

A function is made abstract by the *abstract* directive. An abstract function cannot be defined in the class of its origin as it is absurd. The purpose of abstract functions is to declare an interface that must be defined by the derived classes. The derived classes must use the *virtual* directive for the implemented abstract function; using the abstract directive is an error. This is a subtle and important distinction between the *virtual* and *abstract* directives with regards to reusing in derived classes.

As mentioned, abstract functions impose an obligation on the derived class to implement them without which the classes can't be instantiated. An abstract function can be anywhere in the hierarchy. This has an interesting and useful consequence. Abstractness can be used enforced anywhere in the hierarchy. A lower base class can re-declare a virtual function implemented by a higher base as an abstract function. This technque is called *reinforcing abstraction*. Note that only virtual functions can be reinforced. Reinforcement is useful mainly in multiple inheritance. It helps set a new semantic starting point

```
TopBase = DemoEnv.Object.CreateClass("TopBase", {
    Functions: {
        TopBaseWork: function(){},
        $virtual$__getName: function(){return "<name from top base>"},
        $virtual$__getID: function(){return "<id from top base>"},
        $abstract$__canWork: function(id){},
        $abstract$__willWork: function(id){},
    },
});

MidBase = DemoEnv.Object.CreateClass("MidBase", {
    CrBaseClass: [TopBase],
    Functions: {
        $virtual$__willWork: function(id){return "yes"},
        $abstract$__canWork: function(id){},
        $abstract$__midbaseCanWork: function(id){},
    }
});

Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [MidBase],
    Functions: {
        $abstract$__midbaseCanWork: function(id){},
        $abstract$__derivedCanWork: function(id){},
    }
});

try{new Derived;} catch(e){}
```

```
JEEP run time error [class Derived]. Instantiation failed due to unimplemented
abstract functions.
    The abstract function 'canWork' declared in base [TopBase] and reinforced in
base [MidBase] is not implemented
    The abstract function 'midbaseCanWork' declared in base [MidBase] and
reinforced in the class is not implemented
    The abstract function 'derivedCanWork' is not implemented
```

Read the error messages carefully. They provide the exact amount of detail regarding the origin and reinforcement of the abstract functions, including the fact that a class reinforced an abstract function rather than inheriting the reinforcement.

6.3.3. Validity

Polymorphism is not enabled inside constructor and destructor. This is conceptually sound and retains the behavior from C++. The reason for disabling is this. The instantiation happens from top down, with a base class being constructed before its derived class. However, polymorphism goes in the opposite direction. If it is enabled in constructor, then calling a virtual function from the constructor will invoke the overridden derived class function, which could be a distant descendent, but that part of the instance is yet to be constructed.

Enabling this behavior will result in either useless calls or abortion (crash in C++). If the derived class checks if its own member variables are valid before using them in a virtual function, then it will never use the variables when invoked from the constructor. Such calls effectively becoming useless as they do nothing. If it doesn't check, it results in accessing non existent properties, thus aborting. Something similar happens with destructor; while in the constructor variables aren't initialized yet, in the destructor they are already destroyed. Note that this problem exists only when variables are constructed in the constructors beyond simple initializations, but since there is no way to know how the variables are instantiated, it is better overall to ban polymorphism inside these special functions.

Disabling means that the call to a virtual function will always invoke the original function. In the case of a derived class it will be the function defined in the base class, the class that introduced the function to the hierarchy. In case of a base class it will be its own function rather then the derived override.

```
let Base = DemoEnv.Object.CreateClass("Base", {
    CONSTRUCTOR: function(){this.Print()},
    Functions: {
        $virtual$__Print: function(){cout("Base.Print")}
    }
});

let Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [Base],
    CONSTRUCTOR: function(){this.Print()},
    Functions: {
        $virtual$__Print: function(){cout("Derived.Print")}
    }
});

new Derived;
```

```
Base.Print
Base.Print
```

The bug that this behavior introduces is one of the nastiest kinds. They fail silently, and often go unnoticed until that evening on Friday when you have planned something special. The bug will be of course be untraceable in the ten minutes you peer into the screen while your ears heat up and mind slowly goes blank. Such pleasant FML evenings can be avoided by using the *`trap-invalid-virtual-call flag`*. As the name states, it traps such invalid calls.

```
Base = DemoEnv.Object.CreateClass("Base", {});
Derived = DemoEnv.Object.CreateClass("Derived", {
    Flags: "trap-invalid-virtual-call",
    CrBaseClass: [Base],
```

```
    CONSTRUCTOR: function(){this.Test()},
    Functions: {$virtual$__Test: function(){cout("virtual Test")}}
});
try{(new Derived).Test()}catch(e){cout(e.message)}

Derived = DemoEnv.Object.CreateClass("Derived", {
    Flags: "trap-invalid-virtual-call",
    CrBaseClass: [Base],
    DESTRUCTOR: function(){this.Test()},
    Functions: {$virtual$__Test: function(){cout("virtual Test")}}
});
try
{
    DemoEnv.Function.ScopedCall(function(){
        Derived.ScopedCreate()
    })
}
catch(e){cout(e.message)}
```

```
JEEP run time error [class Derived]. The virtual function 'Test' was invoked from
the constructor.
The class 'Derived' failed to instantiate.

JEEP run time error [class Derived]. The virtual function 'Test' was invoked from
the destructor.
```

When trapped from the constructor the instantiation is aborted. This is the only correct action that can be taken. It aborts during destruction too. This might cause resource leak, but the idea is to trap such things in development mode. And as we know, resource is cheap in development mode, so it must not be an issue.

This flag is useful when classes make member function calls from constructor or destructor. In a class of moderate complexity, it becomes hard to track all functions that call virtual functions, especially when the implementation is under development or refactoring. This flag can be a life saver. Further, it helps trap a particularly perverse kind of invalid virtual call, one that will evade even the keenest of the hawk eyes. Consider this example.

```
Derived = DemoEnv.Object.CreateClass("Derived", {
    Flags: "trap-invalid-virtual-call",
    CrBaseClass: [Base],
    CONSTRUCTOR: function(callback){
        callback(this)
    },
    Functions: {
        $virtual$__Test: function(){
            cout("virtual Test")
        }
    }
});
try
{
    new Derived(function(c){
        c.Test()
    })
}
catch(e){cout(e.message)}
```

7. Multiple Inheritance

Multiple inheritance is a technique of using more than one base class. There is no equivalent to this in JavaScript. and what Jeep provides is completely new. Conceptually, multiple inheritance is an extension of single inheritance, but with additional constraints and variation some behavior. Everything said about single inheritance in the previous chapter holds, so in this chapter I will not repeat it and focus only on the differences. Having read the previous chapter is essential to understand this chapter.

Syntactically, it is quite trivial to use multiple inheritance – simply list all the base classes. In most languages that support multiple inheritance, the syntax is equally trivial. Though trivial to incorporate, multiple inheritance has far reaching semantic and behavioral consequences. So it is not surprising that it has quite a bad reputation in software engineering. The main reason for the bad reputation is the lack of understanding the fundamental difference between multiple inheritance and single inheritance. It is so rampant that people claim composition is better and preferable as a blanket statement. I will say more about this later in the chapter.

7.1. Constructor and Destructor

The order of listing the base classes determine the order of the calls to constructor and destructor. The example shows a very complex inheritance shape to drive the point home. It is what is known as the *diamond inheritance,* where the base classes themselves have a common base. The example further complicates the shape by adding an unrelated base class to the mix. I call this *the diamond and shaft inheritance*.

```
let TopBase = DemoEnv.Object.CreateClass("TopBase", {
    CONSTRUCTOR: function(){cout("TopBase.CONSTRUCTOR")},
    DESTRUCTOR: function(){cout("TopBase.DESTRUCTOR")},
})

let MidBaseA = DemoEnv.Object.CreateClass("MidBaseA", {
    CrBaseClass: [TopBase],
    CONSTRUCTOR: function(){cout("MidBaseA.CONSTRUCTOR")},
    DESTRUCTOR: function(){cout("MidBaseA.DESTRUCTOR")},
})

let MidBaseB = DemoEnv.Object.CreateClass("MidBaseB", {
    CrBaseClass: [TopBase],
    CONSTRUCTOR: function(){cout("MidBaseB.CONSTRUCTOR")},
    DESTRUCTOR: function(){cout("MidBaseB.DESTRUCTOR")},
})

let BaseX = DemoEnv.Object.CreateClass("BaseX", {
    CONSTRUCTOR: function(){cout("BaseX.CONSTRUCTOR")},
    DESTRUCTOR: function(){cout("BaseX.DESTRUCTOR")},
})
let Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [MidBaseA, MidBaseB, BaseX],
    CONSTRUCTOR: function(){cout("Derived.CONSTRUCTOR")},
    DESTRUCTOR: function(){cout("Derived.DESTRUCTOR")},
})

DemoEnv.Function.ScopedCall(function(){
    Derived.ScopedCreate();
})
```

```
TopBase.CONSTRUCTOR
MidBaseA.CONSTRUCTOR
MidBaseB.CONSTRUCTOR
BaseX.CONSTRUCTOR
Derived.CONSTRUCTOR
Derived.DESTRUCTOR
BaseX.DESTRUCTOR
MidBaseB.DESTRUCTOR
MidBaseA.DESTRUCTOR
TopBase.DESTRUCTOR
```

Notice that the diamond part that begins with the *TopBase* still gets constructed in the expected order. Had *BaseX* been listed first, it would have been called before *TopBase*, but the *MidBaseA* and *MidBaseB* would still come after *TopBase*. Essentially, the order of construction of single inheritance is repeated for every line of inheritance.

When construction fails, the line of inheritance that failed will behave exactly as in single inheritance failure. If other lines were constructed prior to the failure, the destructors would be called on all those lines as it would be during a valid destruction of a single inheritance. This example extends the previous one.

```
MidBaseA = DemoEnv.Object.CreateClass("MidBaseA", {
    CrBaseClass: [TopBase],
    CONSTRUCTOR: function(){throw 'MidBaseA exception'},
})

let BaseY = DemoEnv.Object.CreateClass("BaseX", {
    CrBaseClass: [BaseX],
    CONSTRUCTOR: function(){cout("BaseY.CONSTRUCTOR")},
    DESTRUCTOR: function(){cout("BaseY.DESTRUCTOR")},
})

Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [BaseY, MidBaseA, MidBaseB],
    CONSTRUCTOR: function(){cout("Derived.CONSTRUCTOR")},
    DESTRUCTOR: function(){cout("Derived.DESTRUCTOR")},
})

try
{
    DemoEnv.Function.ScopedCall(function(){
        Derived.ScopedCreate();
    })
}catch(e){cout(e.message)}
```

```
BaseX.CONSTRUCTOR
BaseY.CONSTRUCTOR
TopBase.CONSTRUCTOR
TopBase.DESTRUCTOR
BaseY.DESTRUCTOR
BaseX.DESTRUCTOR
```

7.2. Reinforcement of Abstraction

To review, reinforcement of abstraction is a technique where one of the base classes in the hierarchy declares an inherited virtual function as abstract to set a new semantic starting point for its descendents.

It is a useful and powerful technique, but can get undermined in multiple inheritance. If you imagine inheritance as a pipe through which virtual functions from base class flow to derived classes, reinforcement is like blocking the pipe. Now, if there are two base classes such that one reinforces and the other doesn't, you can see that reinforcement has no effect. So Jeep simply establishes a rule – reinforcement stays even if one base does it.

```
TopBase = DemoEnv.Object.CreateClass("TopBase", {
    Functions: {$virtual$__SomeFunction: function(){}}
})

MidBaseA = DemoEnv.Object.CreateClass("MidBaseA", {
    CrBaseClass: [TopBase],
})

MidBaseB = DemoEnv.Object.CreateClass("MidBaseB", {
    CrBaseClass: [TopBase],
    Functions: {$abstract$__SomeFunction: function(){}}
})

Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [MidBaseA, MidBaseB],
})

try{new Derived}catch(e){cout(e.message)}
```

```
JEEP run time error [class Derived]. Instantiation failed due to unimplemented
abstract functions.
    The abstract function 'SomeFunction' declared in base [TopBase] and reinforced
in base [MidBaseB] is not implemented
JEEP aborted.
```

7.3. The Problems and Solutions

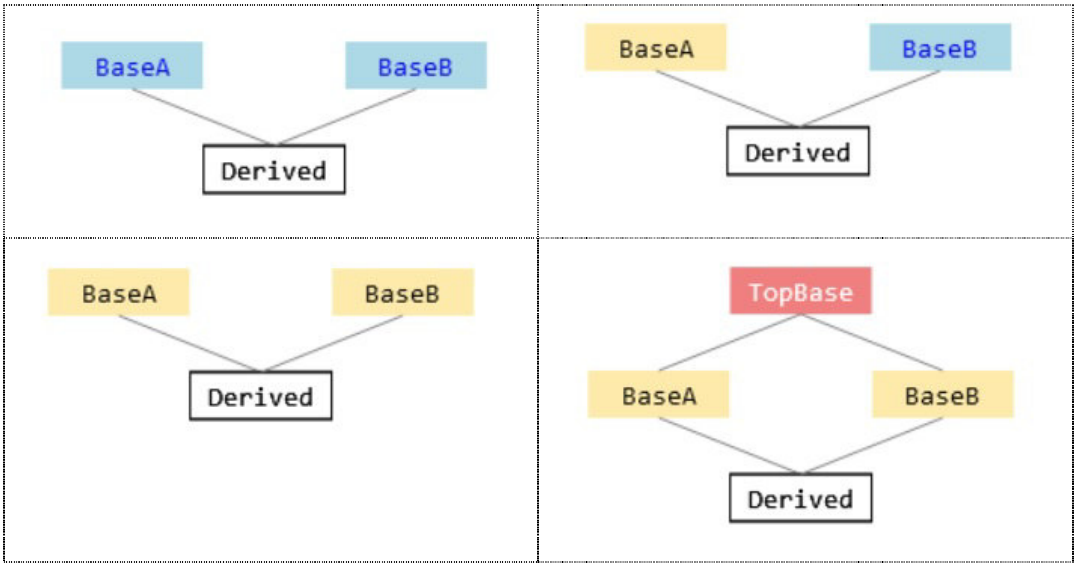
As with any topic, people who dislike multiple inheritance put a lot of subjectivity into it. But there is a valid objective problem too, that of dealing with duplicate names. What happens when more than one base has the same variable name or function name? The base itself might have multiple bases and have inherited them rather than having introduced them. Different programming languages have different solutions of varying amount of validity and soundness. Jeep tries to give its own, which is a bit different from what is usually done (as far I know).

7.3.1. Rule Based Solution

Jeep already establishes the ground rule that a public or a protected member name can appear only once in the hierarchy. While this was beneficial in single inheritance, it is crucial in multiple inheritance. This rule solves most of the problem single handedly. It applies to both virtual and non virtual functions

Consider these hierarchies. They are screen shots of a simple hierarchy visualizer tool that is part of the demo examples. It is of course created using Jeep. The diagram is colored with

respect to some assumed member function, say *SomeFunction*. The red shade indicates it is abstract, yellow indicates it is implemented, gray indicates it must be implemented, blue indicates plain member. White indicates absence of the function.



The top left one with both blue is invalid due to the ground rule. The top right is invalid because a function cannot be both virtual and non virtual. Duality of this kind is primarily a semantic error, which causes undefined behavior or crash. A function in *BaseB* that calls *SomeFunction* expects the implementation present in its own class to run, but something else would run if duality is allowed.

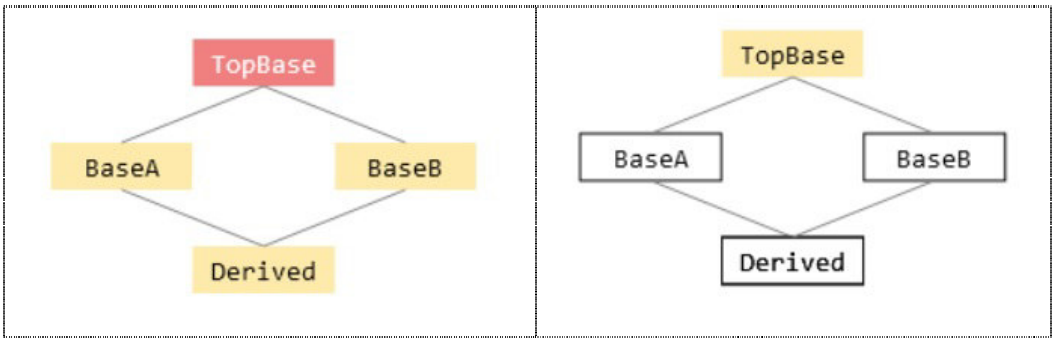
It gets a bit complex with clashing virtual functions. Jeep recognizes two types of virtual functions in a hierarchy, one having same source and one having multiple sources. Such a hierarchy is depicted by the bottom left and bottom right diagrams respectively.

As with duality, it is an error for the same function to be virtual in multiple bases. While both are virtual, unlike duality, there is a still a semantic error. Just because *BaseA* and *BaseB* both have a virtual function with same name, it doesn't mean the function has the same meaning or is used in the same context in both the classes.

When there are clashing virtual functions originating from same source, the error is due to ambiguity rather than semantics. There is no way to choose which virtual function must be used for the derived class. Different languages offer different solutions (I don't recall the names of the languages, and I am lazy to google, so would use *some* instead) Some call all virtual functions in the order of the base classes listed, which is just absurd. Some arbitrarily choose the function present in the left most or right most base without any application based reasoning, which is worse and absurd. The second method is worse because it is prone to, what I call, *the last base bias problem*. The left most and right most choices are essentially a product of how base classes are listed in the class description. Something as important and crucial as polymorphism ends up at the mercy of how a programmer mentions the bases. This method is biased either towards or against the last base. There can't be a worse solution.

The only correct way is to let the programmer choose which functions must the derived class use. He might end up doing one of the things mentioned above, but that is a thing chosen by the programmer and not a auto generated solution with a one-size-fits-all approach. This is what Jeep enforces.

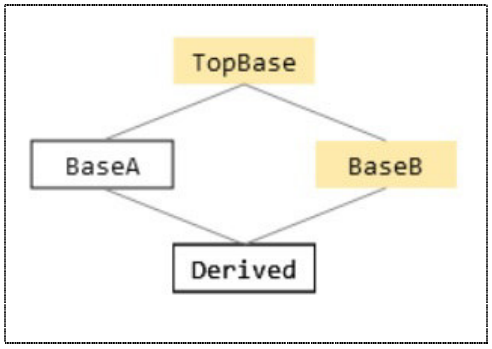
The following scenarios are valid.



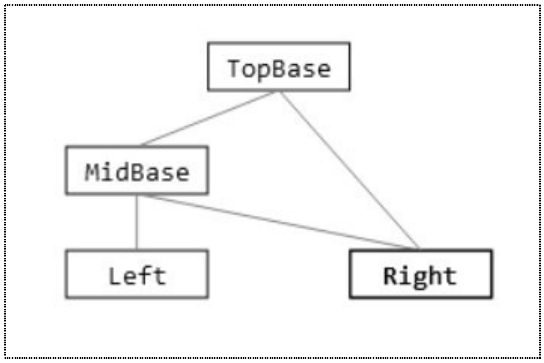
The left one is how ambiguity resolution looks like. The derived class implements the function inside which it might just forward the call to one or both bases, or do something else.

In the right one, both *BaseA* and *BaseB* inherit the same function which is inherited by the *Derived*. There is no real cash because the derived class receives the same function from multiple sources.

However, the following structure is invalid because there are still two clashing virtual functions from the same origin. The *BaseA* simply inherits from *TopBase* and *BaseB* implements its own. It is a variation of the bottom right above.

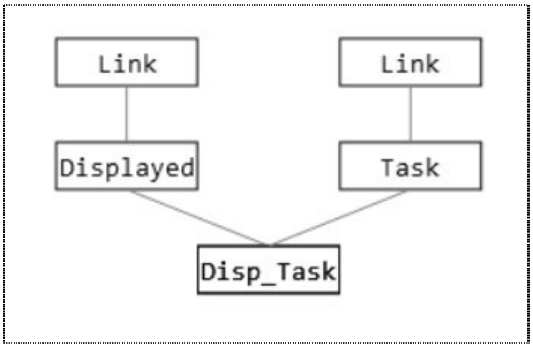


The following structure is also invalid in Jeep due to the derived class *Right*, though it is valid in C++, and probably other languages. Note that all boxes are white, meaning this structure is invalid even if none of the classes have *SomeFunction*.



I see no reason why this shape must be allowed. Due to inheritance, whatever is in *TopBase* is available to *MidBase* and its descendents. There is no difference in what *Left* inherits and what *Right* inherits. So, deriving directly from a base as well as an ancestor seems superfluous and unsound. Its validity might be arguable only if a language allows a derived class to delete members it inherits from its base classes such that its descendents don't get them. But such a language opens a different can of worms, and a triangle inheritance just makes a bad case worse.

Due to these rules, a hierarchy like the following is also invalid in Jeep but quite acceptable in C++. This is an example taken directly from the book *The Design and Evolution of C++*. Even Stroustrup himself begrudgingly allows the pattern to exist.



The *Link* is supposed to implement a linked list, and every class that derives from it is supposed to become a node in the list and get a *Next* and *Prev* functions to iterate. I have used such a shape in my C++ code but only for single inheritance. It avoids using a rather elaborate C++ style container and an iterator for access, and allows quick and simple C style access. I don't see any other valid use cases for this shape. A C++ style is always preferable unless the access is done in a highly time critical loop on a memory constrained system, in which case the C style is better. Further, this is one scenario where composition is the natural, preferable and correct solution when multiple inheritance is involved.

Apart from the utility of this shape, the reason it is invalid in Jeep is due to the nature of JavaScript. For this pattern to be useful, each derived class must be its own list, with its own set of variables that help in iteration. For this, each derived class of *Link* must get a separate copy of the *Link* variables. This part is built into C++ language by design, so it is acceptable there. In contrast, in JavaScript the classes are always merged into one. So there exists only one set of *Link* variables in the instance. Mimicking the C++ way isn't impossible, but doing it will lead to unnatural and cumbersome code in the derived classes.

As with the *Disp_Task* example, one reason why many inheritance patterns are invalid is due to the need to merge the classes. A base class loses its identity once it is merged. With members having one to one mapping with names, there is no way to accommodate clashing members. You might feel such members could be moved to the `$base` property and allow C++ like disambiguation. But that doesn't solve the semantic problem, the another reason. If in one base the virtual function *Print* is meant to be pure output and in other base it is meant to update a member variable for processing by the base, what should the class that derives from both do that is meaningful when it disambiguates? The same problem exists even with non virtual clashing functions, and in general with all clashing members. Fixing the syntax and adjusting the structure is futile if semantics fail.

7.3.2. Wrapper Based Solution

Jeep is rather strict in admitting classes into a hierarchy. This aspect, quite counter intuitively, harms the adoption of multiple inheritance. Usually, multiple inheritance is undertaken with classes from different sources. Not every class that becomes a base might come from the same programmer, same team or same company. So it is to be expected that not all classes would be deemed eligible by Jeep to be in the hierarchy. With Jeep being such an authoritarian, most classes would fail to be useful to be bases; thus harming the adoption of multiple inheritance.

The previous section shows a variety of invalid cases, but there is a common theme in all of those. The problem can be reduced to a problem of managing duplicate names. So, if we rename the functions such that the names won't clash, ineligible classes can be admitted to the hierarchy. But it would be foolish to have the source code of the classes changed just to use a feature of a framework. And these classes might be from third party library, making the name change undesirable. Further, the name clash really depends on hierarchy the class is included to. It would be foolish beyond comprehension to have multiple copies of the same class with different function names to suit different hierarchies. So there must be a solution to this problem that considers all these points.

This problem is actually a ramification of the rule based solution to the fundamental problem of multiple inheritance. But it is a problem worthy of having and solving in order to build semantically correct and robust software. The best part of rule based solution is that it allows understanding the hierarchy, its behavior and even its inner working by simply inspecting a hierarchy diagram as it guarantees that no ambiguity of any kind can exist. Otherwise, the only way to be sure would be to read the code carefully. So this problem is a worthy one.

The solution is provided by class wrappers. They rename the clashing functions and variables.

Of all features, wrappers have by far have the simplest descriptor object but are the most complex in terms of behavior and implementation, so the demonstration must be done in multiple steps. The behavior is more about not doing the wrong thing, so it must be shown that wrappers behave transparently no matter what kind of classes are wrapped.

The class to be wrapped is called the *wrapped class*. The result of wrapping a class is a class wrapper or a *wrapper class*. When you see a class causes name clashes, you create a wrapper class for that and use it in place of the wrapped class as the base class. The following example is Wrapper 101, but still shows a lot of detail. So consider it for a while.

```
DemoEnv.Object.RegisterClass("WrDemoBase", {
    Functions: {
        Print: function(){
            cout(this.$name+".value: "+this.value)
            cout(this.$name+".count: "+this.count)
        },
        Change: function(v){
            cout(this.$name+".Change")
            this.value = v;
        },
        CallChange: function(v){this.Change(v)},
    },
    Variables: {value: 0, count: 33},
});

let Wrapper = DemoEnv.Object.CreateWrapperClass("Wrapper", {
    Class: "WrDemoBase",
```

```
    Functions: {Change: "Modify"},
    Variables: {value: "val"}
});

let Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [Wrapper]
})

let d = new Derived;
d.CallChange(100);
d.Print();
d.Modify(-1);
d.Print();
d.val = -2;
d.Print();
```

```
WrDemoBase.Change
WrDemoBase.value: 100
WrDemoBase.count: 33
WrDemoBase.Change
WrDemoBase.value: -1
WrDemoBase.count: 33
WrDemoBase.value: -2
WrDemoBase.count: 33
```

The function *`CreateWrapperClass`* present in the *`Object`* does the wrapping. Its syntax is consistent with other API but has different semantics. The descriptor has three properties. The class to be wrapped is indicated by the *`Class`* property. The properties *`Functions`* and *`Variables`* are a map of old name and new name.

Wrapping works only with registered classes because it is absurd to wrap created classes. The point of wrapping is to use an otherwise inadmissible class in a hierarchy without having to change its code. Created classes tend to be local or ad hoc, so their code can be changed if necessary.

Notice that unlike inheritance code shown so far, the wrapped class uses *`this.$name`* and it prints the base class name instead of the derived class name. This is because the wrapper doesn't quite merge the classes to form one entity. I will talk about this briefly in ***Internals*** chapter.

After wrapping, to the derived class and its descendents, the old names are replaced with the new ones. The old names cease to exist and any attempt to reference them is an error. This snippet demonstrates that. Notice that the members that are renamed are effectively (and actually) deleted form the object, but those that are not renamed remain intact.

```
cout("Derived.prototype.Change " + (Derived.prototype.Change ?
                                                    "present" : "absent"));
cout("Derived.prototype.CallChange " + (Derived.prototype.CallChange ?
                                                    "present" : "absent"));
cout("d.value " + (undefined!=d.value ? "present" : "absent"));
cout("d.count " + (undefined!=d.count ? "present" : "absent"));
```

```
Derived.prototype.Change absent
Derived.prototype.CallChange present
d.value absent
d.count present
```

Wrapper classes are not registered because they are highly class and hierarchy specific and not reusable. They are meant to be immediately disposed once the derived class is generated. For the same reason, wrapper classes cannot be instantiated. Any attempt to do so will abort the script. Wrapping works with one class only. So if there are other offending classes, each one needs its own wrapper. It is a judgment call to decide which class needs to be wrapped. It has no bearing on the hierarchy.

Wrapping faces with a very subtle and complex problem. Notice that though the derived class uses new names, the wrapped class itself uses the old names, and it works. It solves the underlying problem that I call T*he FGK Problem*. It is when you rename the member F to K but there exists another member G that references F. The renaming of members and the FGK problem are in direct opposition to each other. Wrapper solves the tension such that preexisting classes remain untouched.

Further, wrappers maintain the robustness of public and protected functions if present. They also maintain the polymorphic behavior when virtual functions are wrapped. In essence, apart from the renaming, everything else about the wrapped class remains intact.

Any class can be wrapped, even a wrapper class. The mechanism doesn't (and can't) check if the wrapping is necessary. Wrapping a wrapper is just a wasteful effort and obviously not. Note that the generated wrapper must be registered first before it can be wrapped.

Before discussing more about wrappers, let me show you a use case.

Consider this rather extreme example of a class having to derive from two classes that have exactly the same function and variable names, and another class that is partially the same.

```
let commonDesc = {
    Functions: {
        Print: function(){
            cout(this.$name+".value: "+this.value)
        },
        Change: function(v){
            this.value = v;
            this.Print();
        },
        CallChange: function(v){this.Change(v)},
    },
    Variables: {value: 0},
}

DemoEnv.Object.RegisterClass("WrMIBaseA", JEEP.Utils.ShallowClone(commonDesc));
let BaseAW = DemoEnv.Object.CreateWrapperClass("BaseAWrapper", {
    Class: "WrMIBaseA",
    Functions: {Print: "PrintA", Change: "ChangeA", CallChange: "CallChangeA"},
    Variables: {value: "avalue"}
});

DemoEnv.Object.RegisterClass("WrMIBaseB", JEEP.Utils.ShallowClone(commonDesc));
let BaseBW = DemoEnv.Object.CreateWrapperClass("BaseBWrapper", {
    Class: "WrMIBaseB",
    Functions: {Print: "PrintB", Change: "ChangeB", CallChange: "CallChangeB"},
    Variables: {value: "bvalue"}
});

DemoEnv.Object.RegisterClass("WrBaseX", {
    Variables: {value: 29},
```

```
    Functions: {
        Print: function(){cout("BaseX.value: "+this.value)}
    }
})

let Derived = DemoEnv.Object.CreateClass("Derived", {
    BaseClass: "WrBaseX",
    CrBaseClass: [BaseAW, BaseBW],
});

d = new Derived;
// test BaseA
d.CallChangeA(100)
d.Print()
d.PrintB();
d.ChangeA(200)
d.avalue = -1;
d.PrintA();
// test BaseB
d.CallChangeB(300)
d.Print()
d.PrintA()
d.ChangeB(400)
d.bvalue = -2;
d.PrintB();
```

```
WrMIBaseA.value: 100
BaseX.value: 29
WrMIBaseB.value: 0
WrMIBaseA.value: 200
WrMIBaseA.value: -1
WrMIBaseB.value: 300
BaseX.value: 29
WrMIBaseA.value: -1
WrMIBaseB.value: 400
WrMIBaseB.value: -2
```

There can be multiple wrappers in a hierarchy. Because wrappers are bona fide classes, it makes no difference if the bases are wrappers or not. The example shows wrappers in single inheritance for simplicity. A multiple inheritance example would be much more cumbersome with much more code. The code is very similar to the previous example, except that the wrappers are stacked up instead of beside each other. It uses the same common descriptor, so I won't repeat it here.

```
DemoEnv.Object.RegisterClass("TopBase", JEEP.Utils.ShallowClone(commonDesc));
let TopBaseWrapper = DemoEnv.Object.CreateWrapperClass("TopBaseWrapper", {
    Class: "TopBase",
    Functions: {
           Change: "ModifyTop",
           CallChange: "CallModifyTop",
           Print: "PrintTop"
    },
    Variables: {value: "topval", count: "topcount"}
});

let mdesc = JEEP.Utils.ShallowClone(commonDesc);
mdesc.CrBaseClass = [TopBaseWrapper]
DemoEnv.Object.RegisterClass("MidBase", mdesc);
let MidBaseWrapper = DemoEnv.Object.CreateWrapperClass("MidBaseWrapper", {
```

```
    Class: "MidBase",
    Functions: {Change: "ModifyMid"},
    Variables: {value: "midval"}
});

let Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [MidBaseWrapper]
})

let d = new Derived;
// test top
d.CallModifyTop(100);
d.PrintTop();
d.ModifyTop(200);
d.PrintTop();
d.topval = -1;
d.PrintTop();
// test mid
d.CallChange(300);
d.Print();
d.ModifyMid(400);
d.Print();
d.midval = -2;
d.Print();
```

```
TopBase.value: 100
TopBase.count: 33
TopBase.value: 200
TopBase.count: 33
TopBase.value: -1
TopBase.count: 33
MidBase.value: 300
MidBase.count: 33
MidBase.value: 400
MidBase.count: 33
MidBase.value: -2
MidBase.count: 33
```

As mentioned earlier, wrappers maintain robustness of all kinds. This example shows constant function only.

```
DemoEnv.Object.RegisterClass("WrRobDemo", {
    Functions: {
        Print: function(){cout("Class.value: "+this.value)},
        CallRead: function(){this.Read()},
        $const$__Read: function(){this.value = 10;},
    },
    Variables: {value: 0},
});

let Wrapper = DemoEnv.Object.CreateWrapperClass("Wrapper", {
    Class: "WrRobDemo",
    Functions: {Read: "NewRead"},
    Variables: {value: "val"}
});

let Class = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [Wrapper],
    Functions: {
        $const$__Read: function(){
```

```
            cout("Derived.Read")
            this.val = 10;
        },
    }
})

let c = new Class;
try{c.CallRead()}catch(e){}
try{c.NewRead()}catch(e){}
c.val = -2;
c.Print();
try{c.Read()}catch(e){}
```

```
JEEP run time error [class Derived]. The function 'Read' is declared constant but
tried to modify the variable 'value'.

JEEP run time error [class Derived]. The function 'Read' is declared constant but
tried to modify the variable 'value'.

Class.value: -2

Derived.Read JEEP run time error [class Derived]. The function 'Read' is declared
constant but tried to modify the variable 'value'.
```

Notice that the names mentioned in the error message are old names even when the error is caused by invoking new names. This oddity occurs in the internal variable change related errors as well. This is one ‘bug’ I am willing to live with be for now. I might fix this in later versions.

As mentioned earlier, wrappers maintain polymorphic behavior. Not only virtual functions remain virtual (and abstract remain abstract), they can also be renamed and the renamed function maintains the original status.

```
DemoEnv.Object.RegisterClass("WrPolyDemo", {
    Functions: {
        CallChange: function(){
            this.printName();
            let v = this.getChangeValue();
            cout("Base value changing to: "+v)
        },
        $virtual$__printName: function(){cout("Base")},
        $virtual$__getChangeValue: function(){return -1},
    },
});

let BaseWrapper = DemoEnv.Object.CreateWrapperClass("BaseWrapper", {
    Class: "WrPolyDemo",
    Functions: {getChangeValue: "GCV"}
});

let Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [BaseWrapper],
    Functions: {
        Test: function(){
        this.CallChange()
        },
        $virtual$__printName: function(){cout("Derived")},
        $virtual$__GCV: function(){return 100},
    },
```

```
});

d = new Derived;
d.Test();
```

```
Derived
Base value changing to: 100
```

The following example shows how the Print function problem described in the previous section can be solved with wrappers. To recap, a class has two bases and both bases have a virtual function called *DoPrint*. One base expects the print to happen as the derived class pleases but in other base the derived class is supposed to set some internal data after processing for the base to process more.

```
DemoEnv.Object.RegisterClass("OutPrinter", {
    Functions: {
        OutPrint: function(what){
            this.$.DoPrint(what)
        },
    },
    Protected: {
        $virtual$__DoPrint: function(what){
            cout("OutPrinter printing "+what+"...")
        }
    }
});

DemoEnv.Object.RegisterClass("InPrinter", {
    Functions: {
        InPrint: function(what){
            this.$.DoPrint(what)
            cout("InPrinter processing data "+this.$.data+"...")
        },
    },
    Protected: {
        data: "",
        $abstract$__DoPrint: function(what){}
    }
});

let InPrinterWrapper = DemoEnv.Object.CreateWrapperClass("InPrinterWrapper", {
    Class: "InPrinter",
    Functions: {DoPrint: "DoInPrint"},
});

let Derived = DemoEnv.Object.CreateClass("Derived", {
    CrBaseClass: [InPrinterWrapper],
    BaseClass: "OutPrinter",
    Protected: {
        $virtual$__DoPrint: function(what){
            cout("Derived printing "+what+ " to screen...")
        },
        $virtual$__DoInPrint: function(what){
            this.$.data = what * 100;
        }
    }
})

let c = new Derived;
```

```
c.OutPrint(-1);
c.InPrint(-1);
```

```
Derived printing -1 to screen...
InPrinter processing data -100...
```

As you can see, wrappers must do a lot of things behind the scenes to make themselves transparent. Due to this they pose performance penalty, but it is noticeable only in very specific scenarios. Generally, wrapping classes with many functions, virtual functions, protected members, and the combinations of these is going to be costly. The cost would be noticed only when instantiating classes having wrapper classes as an ancestor a lot or calling the interface functions thousands of times per second. For low volume it should be acceptable.

If a class is a candidate for performance hit when wrapped, it must first be optimistically included in the hierarchy, and only when it is deemed invalid by the framework should you wrap it. If the wrapped class or one of the other bases in the hierarchy changes the interface, retest the necessity. This might seem like a bit of a chore, but it is better than deciding the necessity based on code inspection and including an otherwise unnecessary wrapper in the hierarchy which might hit the performance.

7.4. Critique

In my opinion, the bad reputation that multiple inheritance has achieved is mainly due to people misunderstanding it. To be fair, the misunderstanding stems from the way this mechanism is implemented in different languages, making this a classic case of bad implementation ruining a good concept. Consequently, most people see multiple inheritance as an evil thing.

Admittedly, there are some appalling applications of multiple inheritance, a case of applying the wrong solution to the wrong problem. But the evil thing perception has become more of a tradition where it is accepted and followed without questioning. And as with traditions, any skepticism is responded in a disparaging tone, laced with a generous dose of contempt, while firmly seated on their high horses.

The usual alternative suggested is composition. It is amusing that many people claim that it is better than inheritance in general. Their main supporting argument is that anything that inheritance can do composition can do as well, so composition can replace inheritance. They further argue that composition is conceptually simpler for developers to understand, thus leading to better code. I have often been tempted to create an account on many forums just to post a response, but a like minded guy would have done it a few posts down.

Their main argument is a slippery slope and regressive; the same argument can be applied to everything. It is somewhat similar to what assembler aficionados used to say. The central problem with composition is that it takes away polymorphism. As a result, if you want polymorphic behavior with composition, there will be enormous performance penalty, which is virtually (ha!) non existent in regular inheritance in comparison.

What if I tell you Jeep allows composition and even does it? Mind = blown? Class wrappers are essentially composition. As discussed, they have a lot of performance overhead, not to

mention a very intricate implementation to get things to work. Composition is not for the faint hearted to read or code, and not for code that must perform well. It is a valid technique of class building, but cannot stand in for inheritance in all cases. Note that wrappers are a workaround, and like any workaround they are less than ideal but quite necessary.

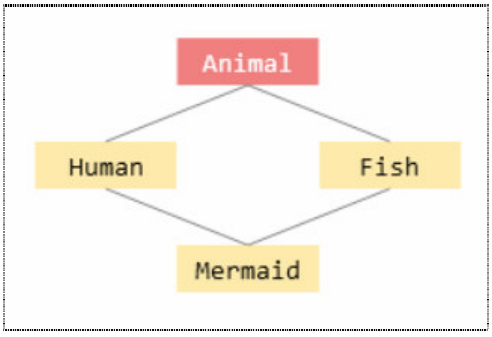
In languages that don't support inheritance, it is hacked using composition. This probably fuels the misconception that it can replace inheritance. While it is somewhat true that most implementation of inheritance is really composition under the hood, it must be left there. Not doing so is like using a number directly because the C/C++ macro uses the same number

As for their other argument, I guess calculus must be reconsidered as a compulsory topic taught since most students just don't get it.

Contrary to the popular opinion, multiple inheritance is the solution and not the problem. It is the solution to the problem of representing things that are a mix of more than one thing. What adds to the bad reputation is people expecting the mixing to happen automatically. In trying to automate it, language designers concoct solutions that are more complex and error prone than multiple inheritance itself. They don't realize that with this technique the programmer must be in full control of choosing how the mixing is done, and any attempt to automate it is doomed to fail. It is one of those cases where programmer intervention is not only necessary but imperative. This is the fundamental mistake people make about multiple inheritance.

A class is composed of properties and action, represented by variables and functions respectively. When mixing is restricted to properties, it can be automated. In fact, automation should be preferred to save a lot of boilerplate code. One such example is representing mixed race people (did I hear you gasp?). Suppose you want to create a person of Indian and Danish descent, and focus only on skin and eye colors. Then, based on popular stereotype, there are many permutations of the cross products of the sets *{brown, white}* and *{black, blue}*. If a *Person* class has functions like *GetSkinColor* (now you gasped I guess), *GetLeftEyeColor*, *GetRightEyeColor* etc, no one would like to write out these functions by hand. This is where automation wins. With Jeep classes, automation means having variables for each of such properties with get and set directives, and using init construction to create persons with specific combinations of traits.

The problem is people expect the same way of automating actions as well. Consider building a game called Angry Mermaids. You would obviously have this hierarchy.



```
DemoEnv.Object.RegisterClass("Animal", {
    Functions: {
        LiveOneMoment: function(){
            cout(this.$name+" living one moment.");
            this.Breathe();
            this.Move(1,1)
        },
        $abstract$__Move: function(x, y){},
        $abstract$__Breathe: function(){},
    }
});
```

```
DemoEnv.Object.RegisterClass("Human", {
    BaseClass: "Animal",
    Functions: {
        $virtual$__Move: function(x, y){cout("  walking to "+x+" "+y)},
        $virtual$__Breathe: function(){cout("  breathing with lungs")},
    }
});
```

```
DemoEnv.Object.RegisterClass("Fish", {
    BaseClass: "Animal",
    Functions: {
        $virtual$__Move: function(x, y){cout("  swimming to "+x+" "+y)},
        $virtual$__Breathe: function(){cout("  breathing with gills")},
    }
});
```

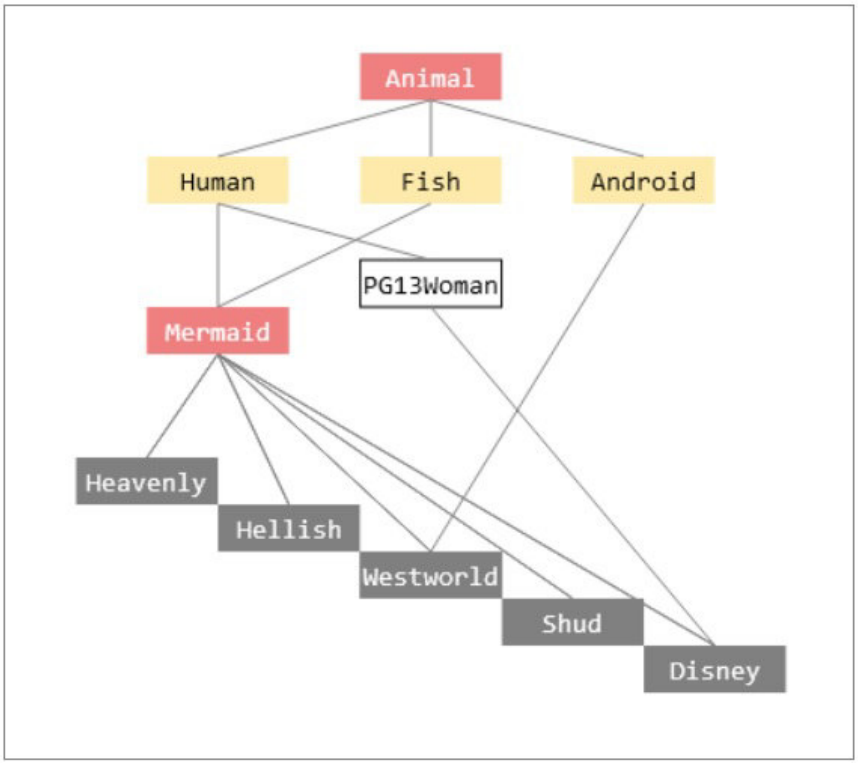
As one might expect, the implementation of *Mermaid* would look like this.

```
DemoEnv.Object.CreateClass("Mermaid", {
    BaseClass: "Human, Fish",
    Functions: {
        $virtual$__Move: function(x, y){this.$base.Fish.Move(x,y)},
        $virtual$__Breathe: function(){this.$base.Human.Breathe()},
    }
});
```

Note that the actions are taken from different aspects of mermaid's nature. This is what I mean by mixing of the actions. There is no automated way to do this. For a simple toy class like this where one function simply calls one other function it seems doable. But we are talking about complex real world scenarios. Most mixing would be a combination of new code, calling helper functions, invocation of overridden base functions etc. All this would often happen inside *if* blocks with application specific logic. Any attempt to automate this with aspects like before, after etc is not going to be useful in any scenarios other than the demonstration code for the system that does this.

In essence, multiple inheritance is a technique where automation must make way for humans.

How about a scenario for some amusement. Let us keep aside our preconceptions and approach the mermaid design with an open mind. It is still a toy, but much closer to the real world design complexity. We might have something like this.



You must notice one thing in the diagram (apart from my attempt at humor and referencing American pop culture). The *Mermaid* class reinforces abstraction. This is done to enforce the open mindedness and come up with different kinds of mermaids that breathe and move differently. The popular perception of a mermaid is the *Heavenly* type. The *Hellish* type will of course be the exact opposite – human movement and fish breathing - because top half would have to be fish. Some might say that *Shud* must be a variety of *Hellish*, but it can't be in this hierarchy.

You might have some code like this with the results as shown below.

```
let mmarr = [
    new(JEEP.GetClass("MMHeavenly")),
    new(JEEP.GetClass("MMHellish")),
    new(JEEP.GetClass("MMWestWorld")),
];
for(let k = 0; k<mmarr.length; k++)
    mmarr[k].LiveOneMoment();
```

```
MMHeavenly living one moment.
  breathing with lungs
  swimming to 1 1
MMHellish living one moment.
  breathing with gills
  holding air for 5 seconds
  standing upright in water
  walking to 1 1
  dunking head back in water
MMWestWorld living one moment.
  <meep morp zeep> <ARNOLD SAYS BREATHE>
                   <activating pump> INTAKING 10 cc air
  <meep morp zeep> <ARNOLD SAYS MOVE>
                   <activating fin motor> STEPPING TO 1 1
```

```
  <meep morp zeep> <ARNOLD SAYS LOOK CAPTIVATING>
                   <activating skin pigment> GLOWING
                   <activating lip motors> POUTING
```

The code for the partial hierarchy might be like this.

```
DemoEnv.Object.RegisterClass("Mermaid", {
    BaseClass: "Human, Fish",
    Functions: {
        $abstract$__Move: function(x, y){},
        $abstract$__Breathe: function(){},
    }
});

DemoEnv.Object.RegisterClass("MMHeavenly", {
    Flags: "need-ancestor-access",
    BaseClass: "Mermaid",
    Functions: {
        $virtual$__Move: function(x, y){this.$base.Fish.Move(x,y)},
        $virtual$__Breathe: function(){this.$base.Human.Breathe()},
    }
});
```

Notice the `need-ancestor-access` flag. It is needed for *MMHeavenly* and others to be able to access the bases of *Mermaid* wherein the implementations lie. This is one of the rare cases where this flag is useful and necessary.

8. Environment and Namespace

The overview chapter provides sufficient information about environment and namespace. Due to being an overview, it glides past many aspects and some are not mentioned at all. This chapter completes the information, acting as a supplement to that chapter, and builds on what is mentioned there. Therefore it is essential to have read that chapter before continuing.

8.1. Environment

As mentioned in the overview, usage of Jeep API starts with creating an environment. The actual API is present in the returned object. There are two aspects of environment that remain to be discussed - client type and production mode flags. The `CreateEnvironment` function would actually look like this most times with three properties (so switching environments might need using slash star star slash instead of to two slashes as was claimed there).

```
AppEnv = JEEP.CreateEnvironment({
    client: “”,
    mode: “development-mode”,
    flags: “”
})
```

The client flags can be either `jeep-aware` or `jeep-agnostic`. They mean what they imply. The created environment object has a function related to the client called `IsClientJeepAware` which returns true or false based on the client flag used. The meaning and necessity of the client type will become clear once production mode flags are understood. As mentioned, there are two kinds of environments you can create, development mode and production mode. The difference between them is the amount of validation, which results in speed differences. To understand the motivation for these modes, you must first understand how Jeep works.

8.1.1. More about Modes

Jeep runs in two phases, the generation phase and the execution phase. The former is essentially the compilation phase of Jeep syntax. It is this phase that has the maximum amount of validation and thus slows things down. Remember that though you understand what Jeep is and how it works, to the client who uses a library that uses Jeep, all that is visible is the slowness of the library they are using. Since Jeep is JavaScript code that processes JavaScript code to produce JavaScript code, there is bound to be delay. It can be several hundreds of milliseconds when there are many classes with complex hierarchies. Even if code can be refactored etc, there aren’t too many different ways in which the validations can be done. So the speed issue will remain.

One solution is to actually produce the plain JavaScript code from Jeep code. It would entail generating an output text that has the class definition as a plain JavaScript code would. Jeep after all has all the information necessary to do this. It certainly appears attractive and novel, but after a moment or two, it becomes evident that it is not scalable. It is also cumbersome to save the output in files after every “compilation”.

The proper solution is to run Jeep in production mode when its client runs on end user machines but in development mode when it is under development. The production mode skips all the generation phase validations. This should not be a problem because generation of classes is a one time thing, no different than adding properties to prototypes in plain JavaScript code. The class and other object descriptions are essentially static with the client having no part in it. If the generation happens without problems in the development mode, there is no reason for it to fail in production mode, save programmer carelessness. So the validation can be skipped to get maximum speed gains. The rudimentary tests shows significant increase in performance, so it is a worthy thing to do.

The execution phase can be divided into two sub phases, the instantiation of objects and execution of object functions. The instantiation phase validation is to see if abstract functions are implemented. The function related validations are the various robustness aspects. These validations do not impact performance, unless used in performance critical points of the application, so it is a judgment call to retain them or not.

By default, production mode doesn't do any validation, including execution phase ones. It must all be explicitly stated via the flags when creating the environment. These flags are valid only for production mode, as all validations are always enabled in development mode. There are several flags, one for each runtime aspect, that must be used to retain that particular aspect. You can of course retain all of them. The flags have *retain* prefixed to their names. The names are pretty self descriptive.

- *retain-const* retains the constantness of a member function
- *retain-argnum* retains the argument count validation
- *retain-argconst* retains the argument constantness
- *retain-argtypes* retains the argument types
- *retain-internal-varchange* retains the effect of the `internal-variable-change` flag
- *retain-invalid-virtual-trap* retains the effect of `trap-invalid-virtual-call` flag
- *retain-abstract-check* retains the validation that disallows instantiating classes with unimplemented abstract functions
- *retain-protected* retains the access violation validation for protected members

This following examples show the same class run once in non retained environment and one in retained environment to show the difference. For robustness, only const directive is used.

The `const` directive

```
function run(mode, env){
    cout("running in "+mode)
    let Class = env.Object.CreateClass("Class", {
        Functions: {
            $const$__Read: function(){this.value = 10;},
            Test: function(){ this.Read(); cout("this.value: "+this.value)}
        },
        Variables: {value: 0},
    });
    let c = new Class;
    try{c.Test()}catch(e){cout(e.message)}
}

run("non-retained production mode", JEEP.CreateEnvironment({
    mode: "production-mode",
    client: "jeep-aware",
}))
```

```
run("retained production mode", JEEP.CreateEnvironment({
    mode: "production-mode",
    client: "jeep-aware",
    flags: "retain-const”,
}))
```

```
running in non-retained production mode
this.value: 10

running in retained production mode
JEEP run time error [class Class]. The function 'Read' is declared constant but
tried to modify the variable 'value'.
JEEP aborted.
```

Validating abstract function implementation

```
function run(mode, env){
    cout("running in "+mode)
    let Class = env.Object.CreateClass("Class", {
        CONSTRUCTOR: function(){cout(this.$name+".CONSTRUCTOR")},
        Functions: {
            $abstract$__Read: function(){},
        },
    });
    new Class;
}

run("non-retained production mode", JEEP.CreateEnvironment({
    mode: "production-mode",
    client: "jeep-aware",
}))

try{
    run("retained production mode", JEEP.CreateEnvironment({
        mode: "production-mode",
        client: "jeep-aware",
        flags: "retain-abstract-check"
    }))
}catch(e){cout(e.message)}
```

```
running in non-retained production mode
Class.CONSTRUCTOR

running in retained production mode
JEEP run time error [class Class]. Instantiation failed due to unimplemented
abstract functions.
    The abstract function 'Read' is not implemented JEEP aborted.
```

Trapping invalid virtual call

```
function run(mode, env){
    cout("running in "+mode)
    let Class = env.Object.CreateClass("Class", {
        Flags: "trap-invalid-virtual-call",
        CONSTRUCTOR: function(){this.Read()},
        Functions: {
            $virtual$__Read: function(){cout("Reading...")},
        },
```

```
    });
    new Class;
}

run("non-retained production mode", JEEP.CreateEnvironment({
    mode: "production-mode",
    client: "jeep-aware",
}))

try
{
    run("retained production mode", JEEP.CreateEnvironment({
        mode: "production-mode",
        client: "jeep-aware",
        flags: "retain-invalid-virtual-trap"
    }))
}
catch(e){cout(e.message)}
```

```
running in non-retained production mode
Reading...

running in retained production mode
JEEP run time error [class Class]. The virtual function 'Read' was invoked from the
constructor.
The class 'Class' failed to instantiate.
```

If the classes you generate are for your own library and not meant to be extended by a client, then there is no need to retain the validations. After all, a millisecond saved is a millisecond gained. However, if there is even a hint of client extension, it is best to retain the validations. Extensions can be made two ways, via derivation or via callback functions given as function arguments. This is where the client type comes into the picture.

8.1.2. Client Type

The client could be either applications that directly use Jeep or libraries that use Jeep. The libraries may or may not expose the Jeep usage to their clients. In any case, creating an environment is an essential step to using Jeep.

As mentioned, the clients of Jeep can themselves use the `IsDevMode` to their advantage and do whatever special processing they want. The jeep aware clients can be expected to use this and be developed in both modes. This means they could be expected to be tested well in their own development phase, where all wrong usages would be caught and fixed. So the production mode validations can be safely skipped.

In contrast, jeep agnostic clients won't follow the two mode system laid out by Jeep. If they do, it won't be using Jeep's environment. It is reasonable to assume that jeep agnostic clients would do things like modify variables directly, invoke functions with wrong arguments etc. For such clients it is necessary to retain validations if you want to provide a robust solution. Having jeep agnostic clients is not that hard to imagine. A library can create classes using Jeep and return the definitions to its clients. Jeep generated classes being normal JavaScript classes, the client would use them as it would any plain JavaScript classes, which includes thoroughly abusing them.

A library can decide how the classes must be generated based on the mode and the client type.

```
function run(env){
    cout("production mode? "+(env.IsDevMode()?"no":"yes"))
    cout("client jeep aware? "+(env.IsClientJeepAware()?"yes":"no"))

    if(!env.IsClientJeepAware())
    {
        env = JEEP.CreateEnvironment({
            mode: "production-mode",
            client: "jeep-agnostic",
            flags: "retain-argnum"
        })
    }

    let Class = env.Object.CreateClass("Class", {
        Functions: {
            $argnum$__Read: function(where, howmuch){
                cout("Reading whare: "+where+" howmuch "+howmuch)
            },
        },
    });
    return Class;
}

let Class = run(JEEP.CreateEnvironment({
    mode: "production-mode",
    client: "jeep-aware",
}))
let c = new Class;
c.Read(1);

try
{
    let Class = run(JEEP.CreateEnvironment({
        mode: "production-mode",
        client: "jeep-agnostic",
    }))
    let c = new Class;
    c.Read(1);
}
catch(e){cout(e.message)}
```

```
production mode? yes
client jeep aware? yes
Reading whare: 1 howmuch undefined

production mode? yes
client jeep aware? no
JEEP run time error [class Class]. The function 'Read' is declared to take 2
arguments but invoked with 1.
JEEP aborted.
```

8.2. Namespace

As mentioned in the overview, a namespace helps avoid name clashes when objects bearing the same name are registered. You could say that having namespaces is a consequence of the registration process that adds object definitions to a common database. While that is true, namespaces offer much more. They help building libraries. The central feature is still name disambiguation, but it manifests differently to be useful in different ways.

As depicted in the overview, a basic library could have this setup.

```
// lib.js
function InitLib(Env, libparams)
{
    Lib = Env.CreateNamespace(“Lib”)
    Lib.RegisterClass(...)
    return Lib
}

// client.js
AppEnv = JEEP.CreateEnvironment(...)
Lib = InitLib(AppEnv, {...})
C = Lib.GetClass(...)
```

8.2.1. Partitioning

Often, a library will have to be divided into subsections for better organization. For that, the init function could create one library per section and return a map of name and library objects to its clients. As you can imagine, it takes a lot of code to do that and leaves a lot of objects to juggle. Because it is an often repeated task, Jeep offers a better solution via an API.

The `CreatePartition` function exists for this very purpose. It is present in the namespace object returned by `CreateNamespace`. The function takes an array of names that represent the partitions.

```
// lib.js
function InitLib(Env)
{
    let Lib = Env.CreateNamespace("Lib");
    let parts = Lib.CreatePartition(["utils", "fileproc"]);

    let desc = {
        CONSTRUCTOR: function(){cout(this.$name)},
        Variables: {value: 0},
    }
    Lib.RegisterStruct("MainStruct", desc)
    Lib.utils.RegisterStruct("UtilsStruct", desc)
    Lib.fileproc.RegisterStruct("FileProcStruct", desc)

    return Lib;
}

// client.js
let Env = JEEP.CreateEnvironment({client: "jeep-aware", mode: "development-mode"})
let lib = InitLib(Env);

let MainStruct = lib.GetStruct("MainStruct")
MainStruct.New();

let UtilsStruct = lib.utils.GetStruct("UtilsStruct")
```

```
UtilsStruct.New();

let FileProcStruct = lib.fileproc.GetStruct("FileProcStruct")
FileProcStruct.New();
```

```
Lib.MainStruct
Lib.utils.UtilsStruct
Lib.fileproc.FileProcStruct
```

The library object on which the function is invoked is divided into the said partitions. This is more practical than creating and returning another object because eventually it is the returned object that will be used. As you can see, you deal with one object only, with all the partitions neatly packed as sub objects. This spares you from juggling with all those library objects that you would otherwise have.

A library can be partitioned any number of times, but it has two basic restrictions. One is that the partition names cannot be the same as the API names. This is because both the partitions and the API reside at the same level of access. The other restriction is that a partition name cannot be repeated for the same library. In development mode, violating either will result in abortion.

8.2.2. Aliasing

Libraries are often built on top of each other for various reasons. One reason is to provide better abstraction. The motivation is usually to provide better interface than the underlying library. Imagine a library *DiskLib* to do low level disk access. It will obviously have a lot of cryptic parameters to do even basic tasks. But applications usually deal with files rather than disks, so a file library *FileLib* built on top of *DiskLib* would be necessary. The application that uses *FileLib* would probably have no idea of the existence of *DiskLib*.

One of the consequences of this kind of library building is that many data structures will have to be translated by the intermediary libraries. However, some structures of the underlying libraries would be so simple that they can be used directly instead of causing code bloat. But this breaks the abstraction. As you can see, maintaining the abstraction and reducing code bloat are in direct opposition to each other. The tension is resolved by aliasing.

Suppose a low level graphics library *LLGL* has a simple record called *Point* with variables x and y. The high level library *HLGL* that builds on *LLGL* creates an alias for the *Point* record instead of recreating it. Its clients use *Point* as if it were *HLGL's* own record, and *HLGL* functions can simply forward the point objects to the *LLGL* functions without needing translations. Any type validation done by *LLGL* functions will admit the forwarded arguments because the *Point* type has been aliased.

The function `CreateAlias` does the aliasing. It takes two arguments. The first is the source namespace object. The second is a descriptor object. The object has three properties – Record, Structure and Class – each being a comma separated string containing the names of the appropriate types of objects. The example only shows aliasing records for brevity.

```
// llgl.js
function InitLLGL(Env)
{
    let Lib = Env.CreateNamespace("LLGL");

    Lib.RegisterRecord("Point", {x: 0, y: 0})
    Lib.RegisterRecord("Color", {base: "black", alpha: 0})
```

```
    Lib.RegisterRecord("Shape", {shape: "round"})

    let Point = Lib.GetRecord("Point")
    let Color = Lib.GetRecord("Color")
    let Shape = Lib.GetRecord("Shape")

    function putPixel(point, color, shape){
        if(Point.InstanceOf(point) == false)
            throw "Expected "+Point.$name;
        if(Color.InstanceOf(color) == false)
            throw "Expected "+Color.$name;
        if(shape && Shape.InstanceOf(shape) == false)
            throw "Expected "+Shape.$name;
        cout("Putting "+color.base+" pixel at x: "+point.x+" y: "+point.y)
    }
    Lib.PutPixel = putPixel;

    return Lib;
}

// hlgl.js
function InitHLGL(Env)
{
    let llgl = InitLLGL(Env);
    let Lib = Env.CreateNamespace("HLGL");
    Lib.CreateAlias(llgl, {Record: "Point, Color"})
    Lib.RegisterRecord("Shape", {shape: "round"})
    Lib.PutPixel = function(p, c, s){llgl.PutPixel(p, c, s)}
    return Lib;
}

// client.js
let Env = JEEP.CreateEnvironment({client: "jeep-aware", mode: "development-mode"})
let Lib = InitHLGL(Env);
let Point = Lib.GetRecord("Point");
let Color = Lib.GetRecord("Color");;
let Shape = Lib.GetRecord("Shape")
let pt = Point.New({x: 10, y: 20});
let clr = Color.New({base: "yellow"});
Lib.PutPixel(pt,clr)
try{Lib.PutPixel(pt,clr, Shape.New())}catch(e){cout(e)}
```

```
Putting yellow pixel at x: 10 y: 20
Expected LLGL.Shape
```

The record *Shape* was not aliased, so it caused the error.

9. Utilities

The framework adds some utilities that applications might find useful. These utilities are not OOP related. Some of them are just functions that help manage some chores of the language while some are Jeep objects that provide new functionality. All the utilities are used in Jeep's implementation, so they are pre tested in real world conditions. They are present in `JEEP.Utils`.

9.1. ObjectIterator

This is structure that provides iterator based solution to accessing object keys and values. The usual method of getting the keys with `Object.keys` uses many variables and array access operator that can get pretty hard to manage, especially when nested This utility provides a nice interface to do the same task. The interface looks like this.

```
ObjectIterator = {
    GetNext: function(){},
    GetCurrPair: function(){},
    GetCurrKey: function(){},
    GetCurrValue: function(){},
    Total: function(){},
    Reset: function(obj){},
}
```

I guess most of the names are self explanatory. The constructor takes the object of interest as the parameter. The `Reset` is present to minimize variable creation and reuse existing ones wherever possible. It takes an object as its parameter, just like the constructor

```
let obj = {a: "A", b: "B", c: "C"}

let iter = JEEP.Utils.ObjectIterator.New(obj)
while(iter.GetNext())
{
    let pair = iter.GetCurrPair();
    cout(pair.key+":"+pair.value)
}

iter.Reset({name: "unknown", location: "undisclosed"})
while(iter.GetNext())
{
    cout(iter.GetCurrKey()+":"+iter.GetCurrValue())
}
```

```
a:A
b:B
c:C

name:unknown
location:undisclosed
```

Notice that this provides a cleaner solution but there is a lot of function calls involved. While this would not harm performance in usual cases (though it is slower than `Object.keys`), it

might harm when used on an object with a large number of keys, or used in a performance critical points of the application. So use your judgment and benchmark before committing to this.

9.2. GetKeyValueArray

This is a function that takes an object and returns an array of key value pairs for the object.

```
let obj = {a: "A", b: "B", c: "C"}

let kvarr = JEEP.Utils.GetKeyValueArray(obj);

for(let k = 0; k<kvarr.length; k++)
    cout(kvarr[k].key+":"+kvarr[k].value)
```

```
a:A
b:B
c:C
```

This is an alternative to using `ObjectIterator` (Jeep doesn't use this). This function has only one call, and is faster in a crude performance test I made. However, this is misleading in my opinion. This solution creates a lot of objects and an array while the `ObjectIterator` structure creates only one object. The array `ObjectIterator` creates internally is an essential thing done even in the usual method, so it doesn't count. When objects have a large number of keys, this act of creation of variables might actually add to the load on the engine in a real world scenario and affect the performance. In contrast, `ObjectIterator` has fixed amount of overhead since the function calls can be expected to have fixed turn around time. So, counter intuitively, `ObjectIterator` might actually be better. All this is pure speculation and not based on measurement. As you can see, there are a lot of engine internal details at play here to decide which to use, so use your judgment and benchmark here as well.

9.3. ForEachKey

As the name implies, this is a high level function that calls a given predicate function for every property present in the given object. The predicate function is passed the object itself and the current pair. The object is passed to reduce closures, dependencies etc.

The function takes an optional third argument which is another function. This function is intended to act as a post processor for the predicate, as it would be called with the return value of the predicate as the third argument. The other two are same as the predicate.

```
JEEP.Utils.ForEachKey(
    {a: 0, b: -1, c: 10},
    function(obj, pair){return pair.value < 0;},
    function(obj, pair, res){
        cout(pair.key+":"+pair.value+" is less than zero? "+(res?"true":"false"))
    },
);
```

```
a:0 is less than zero? false
b:-1 is less than zero? true
c:10 is less than zero? false
```

9.4. ShallowClone and DeepClone

These two functions do as their names suggest. They both take the object to be cloned and return the clone.

In shallow cloning, only the immediate properties present in the source object are copied to the cloned object. If these properties are objects, modifying them in the cloned will reflect in the source. In contrast, deep cloning clones the object recursively. So modifying properties in the clone will not reflect in the source. Due to this, deep cloning is slower than shallow cloning, and is affected by the layout of the object.

```
let Struct = DemoEnv.Object.CreateStruct("Struct", {
    CONSTRUCTOR: function(v){this.obj.value = v},
    Variables: {obj: {value: -1}},
    Functions: {
        Print: function(which){cout(which+" obj.value: "+this.obj.value)},
        ChangeValue: function(v){this.obj.value = v}
    }
})

let s = Struct.New(100)
s.Print("original");
cout("-- shallow clone")
let cs = JEEP.Utils.ShallowClone(s)
cs.ChangeValue(-1)
cs.Print("cloned");
s.Print("original");
cout("-- deep clone")
let ds = JEEP.Utils.DeepClone(s)
ds.ChangeValue(33)
ds.Print("cloned");
s.Print("original");
```

```
original obj.value: 100
-- shallow clone
cloned obj.value: -1
original obj.value: -1
-- deep clone
cloned obj.value: 33
original obj.value: -1
```

9.5. Merge, CopyProps and CopyDefinedProps

These three function help copy properties from one object to another. As you can imagine, Jeep uses these functions quite a lot.

Merge takes two arguments, the source object and the destination object. It simply copies everything from the source to the destination. If destination already has the same names hey will be overwritten, otherwise they are left untouched.

```
obj = {a: "A", b: "B", c: "C"}
cobj = {x: "x", y: "y", z: "z"}
JEEP.Utils.Merge(obj, cobj)
cout(JSON.stringify(cobj))
```

```
{"x":"x","y":"y","z":"z","a":"A","b":"B","c":"C"}
```

CopyProps function takes three arguments – source object, destination object, names array - and copies the properties of the said names from source to destination. If names are not given, it copies everything, effectively merging. It doesn't validate the presence of the property in the source, so if you give invalid names the destination will contain `undefined` against the name. It will also not check if the name is already present in the destination and results in overwriting.

```
let obj = {a: "A", b: "B", c: "C"}
let cobj = {}
JEEP.Utils.CopyProps(obj, cobj, ["a", "c"])
cout(JSON.stringify(cobj))
```

```
{"a":"A","c":"C"}
```

CopyDefinedProps is similar to *CopyProps* in function signature and behavior, except that instead of simply copying property values, it copies property descriptions. This function helps when you have to transfer defined properties from one object to another while keeping the property description intact. *CopyProps* does not retain anything but value.

```
let obj = {}
let value = -1;
Object.defineProperty(obj, "value", {
    enumerable: true,
    configurable: false,
    set: function(v){value=v},
    get: function(){return value},
})

let cobj1 = {}, cobj2 = {};
cout("obj.value: "+obj.value)
JEEP.Utils.CopyProps(obj, cobj1, ["value"])
JEEP.Utils.CopyDefinedProps(obj, cobj2, ["value"])

cout("obj.value (CopyProps): "+cobj1.value)
cout("obj.value (CopyDefinedProps): "+cobj2.value)
obj.value = 100;
cout("obj.value: "+obj.value)
cout("obj.value (CopyProps): "+cobj1.value)
cout("obj.value (CopyDefinedProps): "+cobj2.value)
```

```
obj.value: -1
obj.value (CopyProps): -1
obj.value (CopyDefinedProps): -1
obj.value: 100
obj.value (CopyProps): -1
obj.value (CopyDefinedProps): 100
```

9.6. FlagProcessor

This is a utility meant to ease the task of processing string based flags and creating a number with orable flag codes. This is a structure with a single function *Process*. The constructor takes a map of string to number which is the reference to process the given string.

The *Process* takes an object with three properties. The *flags* property is the input string that contains comma separated names of the flags. Comma is the assumed separator, so using anything else will result in wrong detection of names, and thus errors. The *singleOnly* property is a flag that tells whether multiple flags are allowed in the string. The *markError* property is a flag that tells to create an error list if processing fails due to incorrect flags in the string.

The function returns an object containing the error list and the numeric flag. The error list will be empty if the flags were valid, or *markError* was false and some flags were invalid. It returns null if the *singleOnly* flag is set and there were multiple flags in the string.

```
let F_RED = 1;
let F_BLUE = 2;
let F_GREEN = 4;

let colorFlags = JEEP.Utils.FlagProcessor.New({
    "red": F_RED,
    "blue": F_BLUE,
    "green": F_GREEN,
})

let fs = "red, green"

cout("flag string: '"+fs+"'")
let res = colorFlags.Process({flags: fs})
cout("red present? "+(res.flags & F_RED ? "yes":"no"))
cout("blue present? "+(res.flags & F_BLUE ? "yes":"no"))
cout("green present? "+(res.flags & F_GREEN ? "yes":"no"))

cout("flag string '"+fs+"' with singleOnly true")
res = colorFlags.Process({singleOnly: true, flags: fs})
cout(JSON.stringify(res))

fs = "red, green, black"
cout("flag string '"+fs+"' with markError true")
res = colorFlags.Process({markError: true, flags: fs})
cout("These colors are invalid: "+res.errors.join())
```

```
flag string: 'red, green'
red present? yes
blue present? no
green present? yes

flag string 'red, green' with singleOnly true
null

flag string 'red, green, black' with markError true
These colors are invalid: black
```

This function doesn't care if the same flags are mentioned multiple times as long as they are valid. It is idempotent that way. The order of occurrence of the names also doesn't matter.

With this utility, adding and processing new flags is simple.

1. create a numeric flag,
2. map it to a name,
3. process the numeric flag in the intended location.

The intermediate steps of parsing the text and generating the flag is handled by the structure.

9.7. MessageFormatter

Often, text messages follow the same pattern but differ in detail at only certain places. Such messages can be reduced to *message templates*. A message template contains special tokens called *template arguments* representing the difference in detail. The arguments can be dynamically assigned values, which merge with the rest of the template to generate the message text. Doing this not only avoids repetition of message texts and the inconsistencies it causes, it also makes the message generation easily maintainable and extensible. `MessageFormatter` is a structure to do exactly that kind of processing.

Every template is mapped to an id. The map is used to set up the instance. The structure has only one function `Get` which takes two arguments. The first argument is the message id. The second argument is an object with the map of template arguments and template argument values, both of which are text.

```
let MF = JEEP.Utils.MessageFormatter.New({
    "-say-greetings": "First say -greetings- and then mention the -subject-.",
   "introduce-yourself": "My name is $last-name$, $first-name$ $last-name$.",
    "dollar": "$ means dollar",
    "-arg-delim": "-arg- $arg$",
});

let m = MF.Get("say-greetings", {
    greetings: "Hello",
    subject: "World",
})
cout(m);

m = MF.Get("introduce-yourself", {
    "first-name": "Olya",
    "last-name": "Povlatsky"
})
cout(m);

m = MF.Get("dollar", {
    how: "good",
    what: "World",
})
cout(m);

m = MF.Get("arg-delim", {
    arg: "argument",
    what: "World",
})
cout(m);
```

```
First say Hello and then mention the World.
My name is Povlatsky, Olya Povlatsky.
$ means dollar
argument $arg$
```

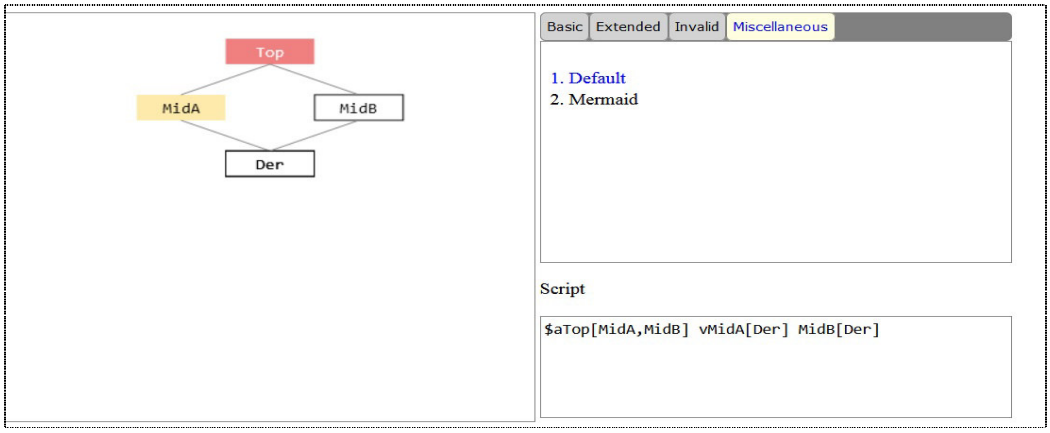
The constructor takes a map of the message id and the message template, both of which are strings. In the template, the argument delimiters must match; the unmatched delimiter is considered part of the template. The message template is processed to identify the arguments automatically. The arguments can appear multiple times in the template. The message ids are not processed, so they can be any string. Actually, the ids are tested only at the first character to check for custom delimiters, but the rest of the string is not processed.

By default, dollar characters are used as argument delimiters. The first letter of the id is assumed to be an English letter. If it is a symbol, it is deemed as the delimiter for the template for that id. This is a quick way to override the default delimiter. This is useful when the message must contain dollar characters in its text and also have arguments (without arguments a single dollar character works). A more structured way is to mention it via the constructor. The constructor takes an optional second argument, a single character, that is used as the argument delimiter instead but it applies to all templates.

10. Examples

It is my belief that no matter how many representative examples are used to showcase a programming system, its true worth will remain under represented unless something real is built with it. By real I don't mean something that is production ready, but rather a toy. A toy doesn't mean a useless concoction. The distinguishing feature of a toy is that it has a general quality of being useful but doesn't consider all scenarios or does validations, yet is attractive enough to be worked on and extended.

The examples I show are the components and libraries used to build the following application. It is a small tool I built to create and visualize the different hierarchies so that I didn't have to swivel at my whiteboard too often. The hierarchies are built using a script. This tool also helped me avoid using tools like MS Paint to draw the structures for the this document and easily generate the shapes I needed. The other reason, and probably the primary reason, I built this was for engineering gratification – there is something profoundly gratifying about building things using the things you have built that assist in building the things that were used to build that thing. I call it *soft bootstrapping*.



Since it is supposed to be a toy, it is a tiny implementation. So all names involved are prefixed with *Tiny*. The code is present in */demo/tiny* of the main location. The application is in the file *tinyhiertree.html*. The two main components are the tree and the tab. The tab simply manages dom elements to achieve the effect. It is present in the `TinyTab` library as a class called `Tab`. The tree, present in the `TinyHierTree` library as a class called `Tree`, uses the `TinyPainter` library which uses the `TinyCanvas` library which ultimately uses the html canvas to do its thing. One of the goals of this example is to show library building and usage.

The script is very crude for obvious reasons and can be improved in obvious ways. The general pattern *is parent[child1, child2...]*. Names are prefixed with dollar or lower case letters, and are assumed to begin with upper case. Boxes and names have one to one correspondence. Using a dot prefixed name gets multiple boxes with same name. The dollar indicates root node; there can be multiple roots The letters a, v, m, q stand for abstract, virtual, member and must implement. The names can be prefixed with minus signs to move the node downwards if things clutter. You may play around with the script but remember that this is a toy and only has limited set of validation and features. For one, circularity is not tested.

The *TinyTab* library has the following setup.

```
function InitTinyTab(Env)
{
    let Lib = Env.CreateNamespace("TinyTab")

    Lib.RegisterRecord("Panel", {
        title: "",
        domElement: null,
    })

    Lib.DeclareClass("Tab", {
        CONSTRUCTOR: function(){},
        Functions: {
            GetDomElement: function(){},
            $AddPanel: "record.TinyTab.Panel",
            AddPanel: function(panel){},
            ActivatePanel: function(pos){},
            UpdateAllTabs: function(){},
            SetSize: function(width, height){},
        },
        Protected: {
            $virtual$__UpdateTab: function(tab){},
        }
    })

    /* implementation details */

    return Lib;
}
```

Notice how useful it is to separate the declaration and the definition of a class. This class is extended by another class to change the tab text color. Notice that in the picture it is blue in one of the tabs. The hierarchy type selected is also set to blue. The blue tab text indicates which tab contains the hierarchy shown. The derived class also does other things like adding and handling the panels. The *GetDomElement* function exists to retrieve and add the dom element to the document body. Notice that protected privilege saves from accidental usage of the virtual function while trying to update all tabs. It was an observation in this example that lead to that feature.

The *TinyCanvas* library has the following setup.

```
function InitTinyCanvasLib(env)
{
    let Lib = env.CreateNamespace("TinyCanvasLib")

    Lib.RegisterStruct("Point", {
        Variables: {
            x: 0,
            y: 0
        },
        Functions: {
            GetDistanceFrom: function(other){
                if(!this.$def.InstanceOf(other))
                    return null;
                return {
                          width: Math.abs(this.x - other.x),
                          height: Math.abs(this.y - other.y)
```

```
                }
            }
        }
    })

    Lib.RegisterRecord("TextProperty", {
        font: "14pt Verdana",
        color: "black",
        align: "center",
        boundHeight: 0,
    })

    Lib.DeclareClass("CanvasXY", {
        CONSTRUCTOR: function(canvas){},
        Functions: {
            GetCanvasElement: function(){},
            Clear: function(){},
            DrawLine: function(start, end, color){},
            DrawBox: function(topleft, rightbottom, color){},
            DrawText: function(at, text, prop){}
        }
    })

    /* implementation details */

    return Lib;
}
```

The name *CanvasXY* suggests it is a two dimensional graphics component. As you can see, it is an extremely component working with just enough things with which the visualizer can be built. However, the interface seems a bit cumbersome, as can be expected from a low level library. Even painting a line is not simple. So, as is the norm, a higher level library, called *TinyPainter*, is created that acts as a broker between the client and this library by providing a much simpler interface.

The *TinyPainter* library has the following setup.

```
function InitTinyPainterLib(env)
{
    let Lib = env.CreateNamespace("TinyPainterLib")

    Lib.RegisterStruct("Line", {
        Variables: {
            xa: 0,
            ya: 0,
            xb: 0,
            yb: 0,
            color: ""
        },
    })
    Lib.RegisterStruct("Rectangle", {
        Variables: {
            x: 0,
            y: 0,
            width: 0,
            height: 0,
            fillColor: "",
            lineColor: "",
        },
    })
    Lib.RegisterStruct("Text", {
```

```
        Variables: {
            x: 0,
            y: 0,
            content: "",
            color: "",
            font: "",
            boundHeight: 0,
        },
    })

    Lib.DeclareClass("Painter", {
        CONSTRUCTOR: function(canvas){},
        Functions: {
            GetCanvasElement: function(){},
            Reset: function(){},
            AddLine: function(line){},
            AddRectangle: function(rect){},
            AddText: function(text){},
        },
    })

    let clib = InitTinyCanvasLib(env);

    /* implementation details */

    return Lib;
}
```

In this approach, each element is self contained which makes things much simpler. Rotating a line, for instance, would be simply changing its location variables. This library uses the `TinyCanvas` library, so it needs to initiate it and use the returned object. But the `TinyCanvas` library is not exported. Thus, it creates an abstraction and the client of this need not know of the `TinyCanvas` library.

The function `AddLine` looks like this. Others follow the same pattern. The class has a member canvas variable to do its business.

```
AddLine: function(line){
    this.canvas.DrawLine(
        this.Point.InitNew({x: line.xa, y: line.ya}),
        this.Point.InitNew({x: line.xb, y: line.yb}),
        line.color,
    )
}
```

The `TinyHierTree` library has the following setup. It demands to be given the painter library rather than initializing it by itself, thus creating an explicit dependency.

```
function InitTinyHierTree(Env, PainterLib)
{
    let Lib = Env.CreateNamespace("TinyHierTree")

    Lib.DeclareClass("Tree", {
        CONSTRUCTOR: function(boundWidth, script){},
        Functions: {
            Paint: function(painter){},
            Reset: function(script, painter){},
        }
```

```
    })

    let Line = PainterLib.GetStruct("Line")
    let Rectangle = PainterLib.GetStruct("Rectangle")
    let Text = PainterLib.GetStruct("Text")

    /* other implementation details */

    return Lib;
}
```

It caches the object definitions instead of access them every time it needs them. These being local to the function pose no threat of polluting the global scope. The tree processes the script and paints the resulting hierarchy. How that is done is not important to mention here.

The way everything comes together is this. The extended tab class has an *`AddTab`* interface function. It takes an array of objects with name and code properties. The function creates an ordered list and adds it as a panel. It also adds a click handler which retrieves the code from the list item and calls `Reset` of the tree. There is only one tree and painter in global scope that everyone accesses. The click handler also calls the *`UpdateAllTabs`* function of which invokes the *`UpdateTab`* which sets the text to blue.

11. Internals

11.1. Implementation

The entire implementation is the *jeep.js* file. The code is written with plenty of white spaces and lots of relevant comments. I believe I have used self descriptive variable and functions names as much as possible. Moreover, I haven't written clever ninja code (I don't ever). Thus, I claim the code can be easily understood by anyone who knows JavaScript. However, a bit of overview helps put things in perspective and spares you having to figure out intricacies. So this section is a very basic overview with most details handwaved. Further, I will avoid listing code as much as possible because code is bound to be refactored, and their listing here would require the document to be updated every time, which is a chore to be honest.

JEEP basically validates syntax and generates classes. When instructed via flags or directives, it generates code that enforce robustness aspects. For this it has to store information. There are various implementation objects scattered through out the JavaScript objects that are produced. They all have the name `_jeepx_` where the 'x' could be something specific, or none at all. As with other underscore members found in regular JavaScript objects, these objects are implementation details and not to be messed with. I followed the naming convention to enforce the established semantic meaning. Note that the exact names, locations and number of these objects are going to change after refactoring.

Jeep is feature rich as you have seen, and it would be neither practical nor useful to discuss implementation of each and every feature. So apart from general syntax validation, I will discuss only a few features that I feel would have piqued your interest.

Before proceeding, I want you to know a few terms that I use in this chapter. Some of these are my concoctions, others are someone else's.

1. *Sandwiching*: This is fairly self descriptive. A function call is bracketed by two sets of code, one for pre processing and one for post processing. The code could be functions.

2. *Layering*: This is when a function processes only some arguments and forwards the rest of it to another function that implement the actual functionality. Actually, the called function could itself be layered. But the implementing function will eventually be called.

3. *Rebasing*: This is where a function is invoked with a specific *this* object. Rebasing happens basically through layering. Rebasing can be done with *call*, *apply* or *bind*. Jeep uses *apply*.

11.1.1. Syntax Validation

The syntax validation is pretty routine and boring. It gets a bit interesting when hierarchies are involved but still pretty routine. The task is split into two functions, `Begin` and `End`, the former doing basic validation and the latter doing more detailed ones. For the declare-define split of the class, two eponymous functions do the task. Each of them does relevant validation before calling the *Begin* and *End* functions. Unlike classes, structures and records have a simple one function implementation.

There are at least two methods in which validation can be done, and I chose the obvious one. The code involves a lot of looping. The loops are mostly $O(n^2)$ but sometimes become $O(n^3)$. For orders above two, I use a map wherever possible as maps are said to be hash tables with

near constant access time. The other method is a mark and sweep approach and can be finished in multiple sweeps, making it O(n). I haven't tried it, and will not, but it surely seems simpler and faster than looping approach. I am not sure of being simpler but I think being faster would be a fallacy. The method involves creating a large array of objects and marking every information possible in every sweep and validating on the last one. This means there would be a lot of objects created, almost one per iteration in the looping method. And it must happen for every generation. It seems this method will produce a lot of garbage and unnecessarily stress the engine. This might be undesirable even in development mode.

All virtual and abstract functions are marked in the *vtable* object present in one of the `_jeepx_` objects. The exact location is going to change shortly while refactoring but this object remains. This vtable is a C++ inspired term. It is a map of name and a record called `VTableRecord`. An entry in this object is the only way to determine if a function is virtual or abstract. All the polymorphism based validations and features are implemented by referring to this object.

The only interesting and somewhat complex parts of code generation are wrappers, and protected and public members. Wrappers are by far the most complex, and it gets more complex when wrapped class has protected members.

11.1.2. Scoping

This is implemented by sandwiching and rebasing. There is a common global stack to which instances are added by the *ScopedCreate* function. The *new* operator doesn't do this. The pre processing marks a new scope. The post processing goes up the stack till the latest mark and calls destructors of all the instances, if the destructors are defined. Each scoped function creates its own mark, so nesting such functions works without a problem. The function `MakeScopedFunction` does the job.

11.1.3. Robustness

All robustness features are implemented by sandwiching and rebasing. A function named *MakeX* implements feature X. Two features need special attention as they do more. They are the ones resulting from the `const` directive and `internal-variable-change` flag.

When these features are present in a class, the object `_jeepsentinel_` is created and added to the instance. It maintains flags that help monitor the access and exception is thrown if violation occurs. The magic lies in the `Object.defineProperty` function. Variables are given set and get property functions that check the sentinel flags, which are manipulated in pre processing and post processing in the functions that implement these features.

11.1.4. Instantiation

Unlike plain JavaScript code where class definition and constructor are the same function, Jeep classes get a special definition function that is different from the class constructor. The function *GenerateConstructor* creates the definition function despite what the name states. The name must be interpreted as *generate internal constructor that calls class constructor*.. The class constructor is always the one provided by the class. The definition function does the instantiation which is a two step process – create an instance and call the constructors with it.

The creation itself is a multi step process, and some of these steps themselves are multi step.

- Check that no abstract functions exist.
- Create variables

- Create protected and private members if necessary
- Create the $*base* property if necessary

When no function is constant or the internal change flag is set, variables are created very similar to how plain JavaScript code creates inside the constructor. If at least one function is constant or the internal variable change flag is set, the variable creation step would create the `_jeepsentinel_` object. The protected and private members result in the creation of the $ and $$ objects. These would have their own sentinels if the instance has one. This is to retain the constantness across all calls.

Since all the functions present in these objects use *this* as if they were public functions, each function is rebased with the instance. The private members being truly private, a new *this* must be created for private functions for rebasing.

At this point, the instance is ready to be 'constructed'. Starting from the root class of the hierarchy the instance class belongs to, constructors of every base class is called if present. Within the loop it is checked if any constructor fails and partial destruction is invoked if necessary. This loop is bracketed by calls to disable and enable polymorphism.

Though the creation of private and protected members is listed in creation phase, it actually happens within the constructor call loop. This is because each base can have its own set of these members and base class access needs looping. It would be wasteful to loop twice.

All this happens for every instance because every instance is supposed to be unique. This is the reason why having overridden virtual functions, robustness, protected members and private members affect the performance when instantiation happens at critical points.

11.1.5. Wrappers

Wrappers are by far the most complex part of Jeep. Behind the rather trivial looking API, and operation that can be described in a sentence, it hides a lot of complexity and intricacy. Its like an unassuming looking nondescript person who has jaw dropping amount of intelligence and knowledge.

To review, wrappers rename variables and function and in the process remove clashing names from the hierarchy. This is quite simple to do – get the object at old name, add it to the new name and delete the old name. But this must still work for the FGK scenario (when a member F is renamed to K, another member referencing G must still work) and retain polymorphism and robustness. Further these three constraints must apply to public and protected members.

The renaming and FGK scenario are in direct opposition. The way it s resolve is via a secret instance (semantically private but I don't want to overload the word) called *wrapped*, which is an instance of the wrapped class. This is done in the constructor of the wrapper class. It is important to note that the *this* in the constructor is the instance of a class that is derived from the wrapper class, but *wrapped* is a local instance of the class that was wrapped. The constructor sets up the function call forwarding mechanism. This needs each function to be rebased with the wrapped object because the *wrapped.G* should be able to refer to F naturally via the `this` object.

The renaming happens as mentioned but on the wrapper class. The wrapper actually creates layered functions that are rebased with the wrapped instance and call corresponding wrapped class functions. After this it renames as mentioned.

If *derived* is an instance of a class derived from the wrapper class, then

derived.newname => wrapped.oldname
derived.unrenamed => wrapped.unrenamed.

For public functions, the task of rebasing is optimized. It is done as it seems less likely that classes choose to use protected members, and then wrap such classes. The optimization involves dynamic linking of the wrapped object. Every function is generated such that rebasing happens during the function call, as with many other features, rather than in the constructor during instantiation. The constructor simply links the created wrapped object which the functions access at a later time to rebase. The overhead of rebasing during instantiation is reduced to simply linking the objects. As a result,. there is a slight overhead in the actual function call as it has to access the wrapped object from a map to rebase. However, since maps are one of the most optimized JavaScript objects, with near O(1) complexity, the overhead can be ignored.

To maintain polymorphism, an intermediate class is created that derives from the wrapped class. This class implements all virtual functions of the wrapped class. The implementation simply forwards all calls to the wrapper class and returns the results returned. This results in calling the derived class implementation if one exists. If it doesn't the wrapped class implementation is invoked due to rebasing mentioned earlier.

To maintain constantness, the wrapped object duplicates the *_jeepsentinel_* if it is present in *this*, or creates one if necessary.

Working with protected members is essentially the same in principle but has some important changes due to the nature of these objects. The main difference is that public members are known at generation time but protected members are created at run time during instantiation. this makes having common code a bit cumbersome and results In extra code.

This is one function that is ripe for refactoring and can be made much simpler than it is.

11.2. Testing

As you know, testing software is a touchy subject for many engineers. We all have strong opinions on various methodologies concocted by committees or a bunch of people at a mega corp. My opinions are so strong that I have concocted my own methodology and a framework, but I won't talk about it here. Not only this is not the right place, it is also something that needs to be discussed at length. There is a C++ framework and an accompanying a 200 page document languishing in a folder since February as I got occupied elsewhere. Though done from C++ point of view, it is equally applicable to any language. The point is, in this section I will not defend or explain why I did the kind of testing that I did but simply state how I did it and what setup I used.

Jeep basically compiles code and generates classes. So it must be tested that wrong code generates errors and right code produces classes. Then, it must be tested that the generated classes have the intended behavior. Further, since Jeep applies many robustness aspects when directives and flags are used, it must be tested that the robustness is in place when is should be. These tests must be done for both development and production modes. That is all there is to it.

Actually there is a bit more. Some utilities are also tested. Though they get tested by virtue of being used in Jeep, not all aspects are used in Jeep, so for the sake of thoroughness, such aspects are tested as well. Many tests such as the syntax validation are not relevant in production mode as such validations are skipped. So such tests are skipped too. And the

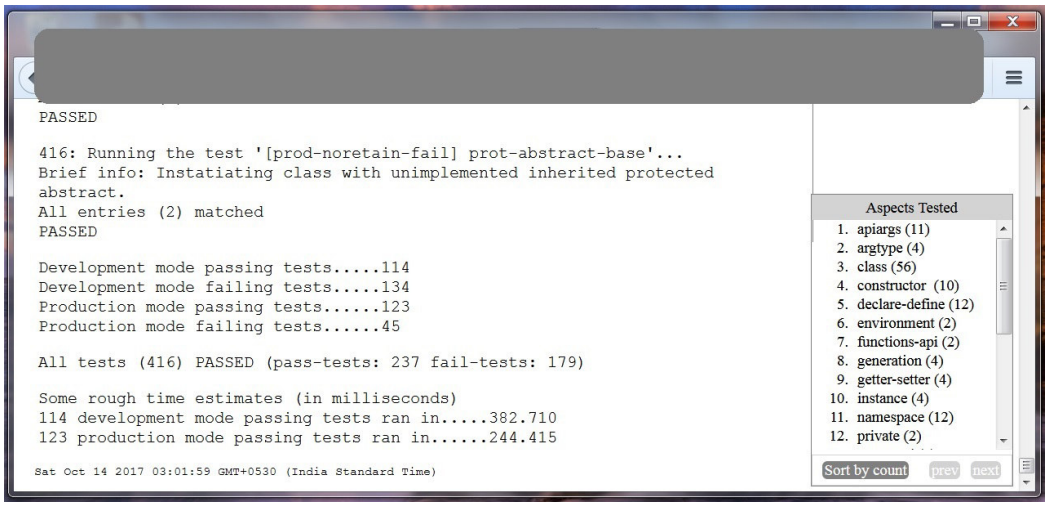
skipping is done by adding the tests only if `IsDevMode` is true. So Jeep is a bit self referential in testing as well. Since this function is trivial, it need not be tested itself etc. I forego that level of pedantry.

Jeep is feature rich, and almost all its features can be combined. As you can imagine, the multidimensional matrix that such a combination produces is quite complex and hard to deal with. However, the fact that most Jeep features are designed to be independent of other features, and implemented accordingly comes to the rescue (I know that the implementation part begs the question but as I said earlier I won't defend anything). Only those features that either conflict or augment each other are tested in combination; others are tested without (actually they are tested in some combination but it is not important to recognize this fact).

The tests are organized based on features. Since no feature can be tested in true isolation as everything is interrelated, and testing one automatically tests some others, no attempt is made to be pedantic in terms of doing things in increasing order of complexity etc. So while a test might be for one particular feature, and that fact is indicated by the name and description, other features would also get tested, whether they are already tested or are yet to be tested.

11.2.1. The Test Framework

A small test framework is created for the purpose of testing Jeep. It is independent of Jeep and is a general purpose code. The basic idea is to add expected results before the test and add generated results during the test; then at the end compare them. The results are all strings. The comparison lists out all the mismatches. When no mismatches happen, the test passes. The results in either case are logged to the html report by dynamically creating dom elements. This is the screenshot of the report it generates.



The framework in implemented in `Tester`, which is a plain JavaScript object with the following interface.

```
Tester = {
    Init: function(){},
    NewCase: function(test){},
    CreateAspectsBrowser: function(){},
    Utils: {}
}
```

The framework needs to be initialized before using. That is done by the `Init` function. The `NewCase` function creates a new test case. The `CreateAspectsBrowser` creates the panel at the bottom right. Its creation is optional but Jeep creates it. There are some utilities in the `Utils` object which do some dom management and formatting.

The test case is implemented as a plain JavaScript class. It has the following interface.

```
Tester.Case = {
    CONSTRUCTOR: function(test){},
    AddGenerated: function(gen){},
    Compare: function(){},
}
```

The argument to the constructor is an object of this format.

```
test = {
    name: “”,
    desc: “”,
    aspects: “”,
    exp: [],
}
```

The `name` identifies the test in the report. They tend to be short and cryptic. The `desc` is supposed to be a longer description for the test to help understand what was done when reading the report. This separation is beneficial when you have to recognize the tests in code. Having a cryptic abbreviated name is much better, even when a long description can be searched. The `aspects` is a comma separated string. Aspects are like tags in blogs. They help recognize which aspects of the component were tested. They help both the developer and the reviewer in assessing the quality of the test. They also allow descriptions to be less wordy and more technical by having the key words moved to aspects. The expected results is an array of text. It is an array because we know before hand all that should happen.

The `AddGenerated` takes a string as argument. This is because, unlike expectation, the act of generating happens one result at a time.

The result of `Compare` is a Boolean value. This function creates the dom elements in the report. The expected and generated results must match exactly, even an untrimmed space in one of them is considered a mismatch.

When created, the aspects browser always sits at bottom right of the report. It lists all the aspects along with the number of tests they appear in. Note that the sum of these numbers would not match the total number of tests because multiple aspects can appear in the same test. The aspects can be sorted based on name or the test count. Upon clicking an aspect, the *next* and *prev* buttons become enabled. They help navigate to the results the aspect appears in.

When tests fail, the report will look a bit different. Both the expected and generated results are listed when there is a mismatch. The failed tests are linked via navigation buttons. All failed tests are all listed at the bottom as well and each entry is linked to the test result.

The failure shown here is the one that gave me several hours of headache and eyestrain. As days went by I began to recognize the pattern and not care to read the message fully. But the time spent dealing with such failures still cost me a couple of days cumulatively. Sadly, I haven’t been able to overcome making it frequently.

```
**  expected : The abstract function 'SomeFunction' declared in base [Base] is not
implemented
**  generated: The abstract function 'SomeFunction' declared in base [Base] is not
implemented.

The test '[prod-noretain-fail] prot-abstract-base' FAILED

3/3 prev next listing

Development mode passing tests.....114
Development mode failing tests.....134
Production mode passing tests......123
Production mode failing tests......45

Total tests: 416 (pass-tests: 237 fail-tests: 179) Passed: 413 Failed: 3

Some rough time estimates (in milliseconds)
114 development mode passing tests ran in.....384.930
123 production mode passing tests ran in......262.500

  1. [dev-fail] prot-abstract-base
  2. [prod-fail] prot-abstract-base
  3. [prod-noretain-fail] prot-abstract-base

Sat Oct 14 2017 03:06:27 GMT+0530 (India Standard Time)
```

11.2.2. The Setup

Jeep uses the above mentioned framework but because it is only for testing, it is not deemed a dependency for clients.

The JEEP object has three functions related to testing. They weren't shown earlier because they are essentially an implementation detail. The function `InitTest` initializes the framework by calling `Tester.Init`. The function `CreateTestCase` create a test case. It does some processing before calling `Tester.NewCase`. Jeep stores the resulting test case because the generated results are yet to be added. The function `RunTestCase` simply calls `Compare` with the stored case

The function `RunJeepTest` present in the *jeeptest.js* does the actual testing. The test cases are roughly organized by features. Most features have passing and failing scenarios. Each feature X is expected to have at least one these functions created inside the *TestEnvInfo* object. Not every feature has both passing and failing scenarios, so such features use whichever function makes sense. The other function will remain undefined and ignored by the testing mechanism. For each feature, an auto generated function invokes these functions.

```
TestEnvInfo.passtest_X = function(env, testList)
{
}

TestEnvInfo.failtest_X = function(env, testList)
{
}
```

These functions are given an environment as an argument so that it can decide which test must be added for which mode. For many features, many development and production mode failures are similar except the syntax validation ones. It is much simpler to put syntax tests inside an `if(env.IsDevMode())` block than separate them into more functions. The other argument is the array into which each function adds its tests.

All tests thus generated are concatenated to get one big list which is run in one big loop. But before that a few things are done. Every test name is prefixed to indicate the mode it is running under. Since most tests run in both modes (protected tests run in three), it would help making sense if a test fails and debug accordingly. Further, the information about whether it is

a passing scenario or a failing scenario is also prefixed. The resulting prefix helps record what kind of tests were done and gives context and meaning to the test results while browsing the report.

11.2.3. Sample Tests

```
testList.push({
    name: "utils-shallow-clone",
    desc: "Tests Utils.ShallowClone",
    aspects: "utils",
    exp: [
    '{"name":"abc","count":1,"sub":{"value":-1,"text":"???"}}',
    '{"name":"abc","count":1,"sub":{"value":-1,"text":"???"}}',
    ],
    func: function(cout){
        let orig = {
            name: "abc",
            count: 1,
            sub: {value: -1, text: "???"}
        }
        let clone = JEEP.Utils.ShallowClone(orig);
        cout(JSON.stringify(orig))
        cout(JSON.stringify(clone))
    }
});
```

```
testList.push({
    //focusThis: TestEnvInfo.SPECIAL_FOCUS_ON,
    name: "prot-access-error",
    desc: "Tests the basic setup of protected members",
    aspects: "protected",
    exp: [
        "JEEP aborting due to run time error. The function 'GetProtValue' of the
                        class [Class] is protected and not accessible directly.",
    ],
    func: function(cout){
        env.Object.RegisterClass("Class", {
            Protected: {
                GetProtValue: function(){cout("GetProtValue")},
            }
        });
        let A = JEEP.GetClass("Class");
        let a = new A(100);
        try{a.GetProtValue()}catch(e){}
    }
});
```

Different tests have different ways of generating the results. The `cout` is a function that ultimately calls `AddGenerated` on the test case. Note that using it always adds to the generated list. So comments are not to be used, unless they help with organizing the expected list. Some tests do this. In that case, the exact message must appear in the expected list, with all the spaces and symbols intact. The test case mechanically compare strings, so its your responsibility to make sure such comment strings match.

By setting the `focusThis` property to `TestEnvInfo.SPECIAL_FOCUS_ON`, some tests get special focus. When even one test gets special focus all other tests that don't get special focus are blocked. Special focus helps with debugging. Without it, breakpoints are triggered by all tests that use the same code path and you have to go through hoops to reach the actual test.

11.3. Performance

Before proceeding, I want you to know that I am not an expert in performance related issues (well, I am not expert in anything). However, like most engineers with my kind of background and experience, I have a fairly good idea of how things work and can indulge in some speculation.

11.3.1. Preamble

Performance testing is quite tricky as a lot of minute details need attention. The environment setup plays a huge part in the test. When doing for JavaScript applications, there is a an added amount of skepticism necessary. It seems to me that the only true way to test performance of JavaScript code is to reset the underlying engine before every test. This is the only way to get an environment that is not susceptible to the effects of cumulative residue left by previous tests. Admittedly, the underlying operating system would be itself susceptible to effects of multiple processes, but stopping at the browser engine seems acceptable and less pedantic. However, such deep dive into the engine is beyond my scope of knowledge and I shall leave it to the experts in the field to pursue if they are interested.

On the flip side, the cumulative residue simulates a real world scenario. Most people would have two digit number of tabs opened in their browsers. Even with a browser that has one process per tab design, the engine would still be common (I guess). So, there is going to be a residue effect. Moreover, with modern operating systems, most of us just close the lid of the laptop and let it hibernate if needed. Restarting happens only when the system gets sluggish. Due to this the browser hardly gets a chance for hard reset. Give all this, it might be not that bad to test without having to restart the engine.

Further, unlike traditional software that run on computers, JavaScript software run on all sorts of devices. For traditional software, there are just a couple of OS that need to be tested for. In contrast, for JavaScript software, there are different versions of browsers running on different versions of different operating systems. And I don't think the browser on our 4096 core smartphone would have one process per tab. The point is, there are a lot more variables (or is it categories?) to consider while talking about performance of JavaScript applications.

This bit about browsers intrigues me. It seems we are back to a runtime environment as was in Windows 3.1 where all processes shared the same memory space. I am not that old to have programmed in Windows 3.1; I just have read about the horrors of working in that setup. A badly written software could penalize every other software in the system, not to mention crash the system. While not that drastic, the way all our JavaScript applications are run in the browser harkens back to that. Most modern browsers try to ameliorate the situation with things like one process per tab. But unless each tab becomes a self contained environment, (essentially multiple tab less browsers), one application will impact other applications.

All these amounts to this: whatever numbers are shown by the performance test are at best very rough estimates, rougher than usual.

11.3.2. The Tests

There are several combinations of factors to test. Jeep's performance

Because Jeep runs in generation phase and execution phase, performance of both phases must be tested. For both phases, the combinations of virtual functions (including overridden), protected and private members, and robustness features apply. And all this with wrappers. Further, the generation must be tested for development and protected modes to see if the split makes a difference. Note that the execution phase test with retained production mode is not necessary. All these results must be tabulated together for comparison etc.

Beyond this, there is a fundamental comparison test that should be done - Jeep with plain JavaScript. And it must be done on all the above combinations that apply to plain JavaScript.

I suspect even a PhD candidate at an A lister university under an A lister professor would test all these combinations if Jeep were his thesis. I certainly don't. It would be too intricate, cumbersome, and honestly quite boring, to both you and me, to include all of them. So I test only a few combinations which seems likely to be used in a real world application. I restrict the tests to these things.

For generation phase I limit to these as they seem to cover all important combinations.

- generating a single level single inheritance
- generating a simple diamond shaped inheritance
- generating a diamond with virtual functions
- generating a diamond with virtual functions and protected and private members

A noteworthy absentee is wrapper class. I intentionally dropped it since the performance hit is evident looking at the implementation. Since wrappers are highly hierarchy dependent, it is better for the test to be done in that context. Further, the numbers shown here are anyway just a rough estimate. So I didn't want to write all that code to test in futility.

For execution phase, I do two things.

- I instantiate all the classes used in generation phase tests. These classes have enough complexity that would affect the instantiation.
- I create a different set of classes to test performance of member variable access when constant functions exist. Since constant read only functions return deep clones, the test is done for simple values as well as a simple object.

Apart from these, I also test the object key access utilities to check how good they are.

The tests are in *jeepperftest.js* in the same location as *jeep.js*. The code depends on the tester framework introduced above. It uses some `Tester.Utils` to create to tabulate the results. All tests were done on my machine with these stats and Firefox 53.0 (64-bit).

11.3.2. The Results

All numbers shown are in milliseconds. Most results are as suspected. Some have rather counter intuitive results, in that they are faster than what is expected to be faster. But that is the effect of the residue and must be ignored.

- Production mode generation is faster than development mode. The difference is going to be wider in real world scenario. Thus, it is beneficial to have the split.
- Having protected and private members affect instantiation speed quite dramatically. However, the drama is noticeable only when hundreds of instances are created. For a handful of instantiations, it is hardly noticeable in the overall scheme of things. So they can be used without apprehension for things like applications, data processors etc that do not have too many instances created and can benefit from semantic guarantees.
- The price paid for deep cloning is also quite significant. As was discussed, unless the code is used by jeep agnostic clients, robustness can be disabled in production mode to get near native speed.

Generation and Instantiation

Class Generation	1	5	10	50
Non Jeep solo	0.075	0.235	0.325	1.480
Development mode solo	1.885	1.800	4.895	11.140
Production mode solo	0.435	0.950	1.795	6.930
Non Jeep hierarchy	0.085	0.190	0.775	2.325
Development mode hierarchy	4.415	4.420	10.015	32.205
Production mode hierarchy	0.965	2.820	8.715	18.885
Development mode diamond	6.165	16.960	24.985	103.335
Production mode diamond	2.195	6.055	10.270	48.220
Development mode virt diamond	9.105	24.965	31.665	106.530
Production mode virt diamond	2.645	9.035	14.345	67.975
Development mode complex diamond	14.350	32.370	62.015	201.840
Production mode complex diamond	4.370	21.930	39.135	152.215

Class Instantiation	1	5	10	50	100	500
Non Jeep solo	0.050	0.050	0.295	0.450	0.555	2.705
Development mode solo	0.360	0.820	1.435	6.975	11.695	55.565
Production mode solo	0.105	0.365	1.060	2.960	6.680	31.770
Non Jeep hierarchy	0.030	0.025	0.100	0.290	0.230	1.130
Development mode hierarchy	0.285	0.940	1.175	5.510	11.560	54.845
Production mode hierarchy	0.100	0.480	0.740	3.260	7.190	33.360
Development mode diamond	0.230	0.875	1.290	5.995	11.690	55.270
Production mode diamond	0.105	0.450	0.735	4.250	7.595	35.745
Development mode virt diamond	0.445	1.165	1.920	10.130	19.245	92.140
Production mode virt diamond	0.180	0.745	1.780	7.230	15.220	72.265
Development mode complex diamond	4.635	8.190	15.210	106.885	232.645	782.005
Production mode complex diamond	1.855	16.385	15.085	55.440	123.110	528.015

- The same classes involved in generation test were instantiated.
- Classes have five empty functions, all public.
- Hierarchy is a single level single inheritance with five empty public functions in both base and derived classes.
- Diamond is a single level diamond with all classes having five empty public functions.
- Virt diamond is same as diamond but with one virtual function in all classes.
- Complex diamond is virt diamond with every class having one constant public, protected and private function each. They also have a public variable with get set, and one protected and private variable.
- Production mode has no flags.
- The columns indicate the number of times the classes were generated, which can be interpreted as being a library with as many classes..

Read and Write

The columns indicate the number of trials.

Value Read Trials	1	10	100	1000	10000
Non Jeep	0.025	0.100	0.235	0.445	1.035
Jeep non constant	0.005	0.080	0.080	0.830	1.195
Jeep constant	0.345	0.745	0.595	4.195	10.340

Value Write Trials	1	10	100	1000	10000
Non Jeep	0.025	0.080	0.085	0.235	2.020
Jeep non constant	0.005	0.125	0.035	0.125	0.855
Jeep constant	0.210	0.505	0.180	1.275	3.120

Object Read Trials	1	10	100	1000	10000
Non Jeep	0.030	0.100	0.065	0.365	1.120
Jeep non constant	0.010	0.080	0.055	0.225	1.060
Jeep constant	0.145	0.235	1.580	6.170	26.020

Object Write Trials	1	10	100	1000	10000
Non Jeep	0.040	0.135	0.085	0.310	1.020
Jeep non constant	0.010	0.115	0.080	0.280	1.060
Jeep constant	0.110	0.230	0.510	4.655	23.685

Object key access

The columns indicate the number of properties in the object.

Object Key Access Trials	10	100	1000
native (Object.keys)	0.070	0.045	0.455
JEEP.Utils.ObjectIterator	0.080	0.100	1.515
JEEP.Utils.GetKeyValueArray	0.210	0.180	0.655

12. Conclusion

12.1. A Review

If you have read the entire document, you would have seen how extensive and complex is Jeep. Had Jeep been created at a mega corp, it would have been be described variously as groundbreaking, paradigm setting and other such hyperbole. I often wonder who comes up with such cringe worthy attributions. But I digress. After reading the document, you would have realized that what Jeep is and what Jeep does could not have been captured in a bulleted list, and the one shown in the features pitch is just the tip of the iceberg. As a consequence, you might be a bit overwhelmed, and wondering how Jeep can be incorporated in your code.

Of all things Jeep does, two are expressly against the grain of JavaScript – validating function argument count, and validating function argument type. The lacunae that these features plug are used in many a code to their advantage. However, doing that is just turning something terrible into something that seems useful, something like the spelling bee competition for the English language. If you think about it, a function is the fundamental unit in the language, and yet so poorly laid out. Jeep fortifies the language from this level. So, even if the OOP part is not useful, these two features alone are good enough to use Jeep.

Similar to C++, Jeep is versatile and has several ways in which to use. It has a pay per use policy, and offers much more than OOP, though OOP is its central feature.

- For code that uses no objects at all but is built around functions only, Jeep provides robustness features that can make the code truly functional rather than being called so just because functions are passed around.

- For code that creates a lot of objects as entries for arrays or key value maps, Jeep helps keep the code clean, consistent and efficient via Records.

- For code that is better written with simple OOP, Jeep offers light weight classes in Structures.

- For code that needs some basic OOP techniques, Jeep offers Class. Even when classes are solitary and not part of a hierarchy, using Jeep classes over plain JavaScript classes makes the code more readable, maintainable, and easily extensible.

- For classes, Jeep expands and improves the concept of constructors, and introduces the concept of destructors. Further, Jeep offers mechanisms of scope creation for automatic invocation of the destructors that help manage resource efficiently.

- Jeep offers three levels of privilege for accessing class members – public, protected and private. The levels are in increasing order of performance overhead and decreasing order of security in terms of being accessible to non member functions. The programmer can pick the right privilege level depending on the application.

- Jeep offers several robustness features applicable to classes which offers semantic guarantees, which is extremely important when writing software of even moderate amount of complexity.

- As for class hierarchies, what Jeep offers is pretty solid. It greatly improves the single inheritance aspect that the JavaScript language has in a very crude way. Further, Jeep offers multiple inheritance that is absent in the language. It also offers simple and elegant solutions to the conceptual problems involved with multiple inheritance.

- Jeep introduces the concept of abstract functions which is crucial in creating interfaces that cannot be used without being implemented.

- Jeep introduces the concept of working in development mode and production mode, where the former is used by the developer and the latter by the end user. For the former, it is acceptable to be slow in order to do validations of all kinds, but the latter is supposed to be as fast as possible. Jeep allows its clients to work in these modes as well for better software engineering.

- Last, but not least, Jeep exposes some very useful code as Utilities that is used in Jeep's own implementation. These utilities will further assist in producing readable, maintainable and extensible code.

After reading through, you might still be wondering if such advanced techniques are needed for software developed in JavaScript. To that, the answer is an emphatic 'no', given that you stick solely to DOM event handlers. For anything other than that, it is an emphatic 'yes'. You might be finding it hard to imagine front end code having such complexities that require something like Jeep. In that case, you are not considering two important scenarios.

One, server side development is increasingly done with JavaScript using frameworks like node.js. Two, desktop development is attempted with JavaScript by using frameworks like Electron. I am sure there will be more frameworks on these lines coming up in the near future. While it is true that you can link to native C++ code from these frameworks to get good performance, thus reducing JavaScript to write just the interlocutor code, the rapid improvements in JavaScript engines make it possible to get near native speed with JavaScript. Additionally, the quick development, deployment and all that can happen with JavaScript compared to traditional languages like C++ makes JavaScript very attractive for serious development.

In general, as with human civilization, until people get the right tools, no one knows what people can create. I hope Jeep will be such a tool that helps people unleash their creativity.

All that said, two questions are pertinent and I try to answer them below.

1. Is Jeep needed now that the JavaScript language is undergoing rapid improvement?

Indeed, JavaScript is improving by the day. But then again, there is hardly anything worthy done OOP wise. There are a few syntactic elements added but they are just sugar to what already existed. Jeep has so much more to offer. So, until the language catches up, Jeep is relevant.

2. What happens when web assembly takes over JavaScript?

While web assembly is good, and brings more developers to web without them having to learn a different language, it is nowhere close to taking over. Even if that day comes before the next ten years, there is no need for JavaScript to stop existing. One can still program in it, as one would with the language that is the fad of the month (and promoted by one of your

friendly mega corps). Admittedly, most people are sort of stuck with JavaScript and will leave at the first chance they get. However, by using Jeep they can overcome many of the troubling features and continue to use a robust version of JavaScript. Why waste all the invested time?

12.2. What's Next?

I think Jeep is complete from usage point of view with no other expressly necessary features pending. It can be considered a public beta and be experimented with. Only implementation side work remains, such as bug fixing, refactoring, performance improvement etc.

However, there are a few things that are interesting enough to be attempted, but still not important to have in Jeep. Some of these are imperfections retained simply because they are not functional bugs and need too much work to correct. If the use cases suggest otherwise, they would be fixed. Some are such features that are garnishes which simply exist to increase the price tags. Others are ambitious hacks to make Jeep more mainstream.

- Fixing the inconsistent names in the error messages with wrapper classes.
- Making just one function argument constant instead of making all of them constant.
- Adding argument type validation and constantness to constructors.
- Expanding argument descriptor to include source of virtual functions.
- Mechanism to convert plain JavaScript class to Jeep class to use them in hierarchies.
- Hacking an open source engine to directly process Jeep on the lines Jeep already does, so that Jeep becomes native to that engine, thereby reducing the framework to a shim!
- Generating meta content for hierarchies to use in visualizing tools like the example.

12.3. Some Last Words

Despite all its faults, I am enamored by JavaScript. It could be a consequence of coming from C++. For one, development is a breeze compared to C++. All I need is literally a notepad and a browser. With browsers being in every device imaginable, I can literally code on my smart TV when rain interrupts play. The only drag factor was the nature of JavaScript. Now that Jeep exists to fortify it, all I have to do is maintain basic coding discipline to reap the benefits.

I am also sold to the premise of node.js. The fact that I can literally move the code across frontend and backend freely is extremely attractive. I did consider ASP.NET but that meant learning C# first. At this stage in my career with one eye on business, learning yet another language and a framework from scratch didn't appeal to me. Make no mistake, I will always remain an engineer at heart, but at this stage time is better spent elsewhere. Now that Jeep exists to allow the sort of code and structure for which C# might have been needed otherwise, I am hooked to JavaScript even more. Note that learning JavaScript was inevitable. Had web assembly been a reality back when I did, I wouldn't have learnt JavaScript for sure.

Now that Jeep is expanded, tested and complete, I am actively *enjeepifying* all my code with much more confidence and rigor than before. Most things I create work best with OOP principles and Jeep has proven to be quite useful to me. JavaScript with Jeep, node.js and C++ is all I need to build my web applications. That is, until my ambitious new programming language and backend framework are ready for production. If you ask why reinvent the wheel, why are they needed etc, I would just say *why not*? For a less facetious answer, you would have to wait till they are published.

ಶುಭಂ

Appendix A

Quick Reference

As the name says, this is a quick reference guide only. Things are only mentioned and briefly explained here. Detailed explanations can be found in the relevant chapters.

A.1. The main object

This is the object that contains the framework's implementation.

```
JEEP  = {
   CreateEnvironment: function(desc){},
   GetRecord: function(name){},
   GetStruct: function(name){},
   GetClass: function(name){},
   SetStdErr: function(printer){},
   Utils: {}
}
```

The descriptor for the CreateEnvironment is as follows.

```
desc = {
   mode: string,
   client: string,
   flags: string.
}
```

- The mode controls how code is generated.
  - *development-mode*: syntax validation is always enabled, other validations are always retained
  - *production-mode*: syntax validation is always disabled, other validations are retained based on the flags
- The client indicates the nature of the client wrt JEEP
  - *jeep-aware*: implies that the client itself might work in the development mode, so production mode validations can be skipped
  - *jeep-agnostic*: implies that the client uses JEEP objects as it would use plain JavaScript objects, so production mode validations might be retained as needed
- The flag is valid for production mode only. It instructs which validations are retained.
  - *retain-const*: retains the constantness of a member function
  - *retain-argnum*: retains the argument count validation
  - *retain-argconst*: retains the argument constantness
  - *retain-argtypes*: retains the argument types
  - *retain-protected*: retains the access violation validation for protected members
  - *retain-internal-varchange*: retains the guard against external modification of member variables, the effect of the internal-variable-change flag
  - *retain-invalid-virtual-trap*: retains the invalid virtual call trap, the effect of the trap-invalid-virtual-call flag
  - *retain-abstract-check*: retains the validation that disallows instantiating classes with unimplemented abstract functions

A.2. The Utils object

These are some functions and structures used in JEEP's own implementation made available for general usage as they are...quite useful.

```
Utils = {
   CopyProps: function(src, dest, names){},
   CopyDefinedProps: function(src, dest, names){},
   Merge: function(src, dest){},
   ShallowClone: function(src){},
   DeepClone: function(src){},
   GetKeyValueArray: function(obj){},
   ForEachKey: function(obj, proc, resProc){},
   ObjectIterator: JEEP.Struct,
   FlagProcessor: JEEP.Struct,
   MessageFormatter: JEEP.Struct,
}
```

A.3. The environment object

This is the result of `CreateEnvironment` present in the JEEP object.

```
Environment = {
   Object: {
      CreateNamespace: function(name){},
      CreateRecord: function(name, desc){},
      CreateStruct: function(name, desc){},
      CreateClass: function(name, desc){},
      RegisterRecord: function(name, desc){},
      RegisterStruct: function(name, desc){},
      RegisterClass: function(name, desc){},
      DeclareClass: function(name, desc){},
      DefineClass: function(name, desc){},
   },
   Function: {
      MakeScoped: function(func){},
      ScopedCall: function(func, args){},
      MakeArgConst: function(func){},
      MakeArgNumValidated: function(func){},
      MakeArgTypeValidated: function(argtypes, func){},
   }
}
```

A.4. The namespace object

This is the result of `CreateNamespace` present in the Environment object.

```
Namespace = {
   CreateAlias: function(src, desc){},
   CreatePartition: function(names){},
   CreateRecord: function(name, desc){},
   CreateStruct: function(name, desc){},
   CreateClass: function(name, desc){},
   RegisterRecord: function(name, desc){},
   RegisterStruct: function(name, desc){},
   RegisterClass: function(name, desc){},
   DeclareClass: function(name, desc){},
   DefineClass: function(name, desc){},
}
```

There are two ways to create record, structure and class definitions. The function *CreateX* returns the definition immediately. The function *RegisterX* adds the definition to a common database. Both functions have the exact syntax and arguments. The first is the name and the second is a descriptor object. Names are alphanumeric, with underscore being the only symbol allowed. The first character of the name must be a letter. For simplicity, only English language letters are allowed.

Registered definitions are accessed using *GetX* functions. The accessor functions present in JEEP search the global database while the ones in the namespace search the database localized by the namespace.

In the following sections, [def] means the definition, [inst] means the instance of the definition and [desc] means relevant contents from the given descriptor.. The name *Generator* is used as a common name for *CreateX* and *RegisterX*.

A.5. Record

```
[def] = Generator(name, desc)
```

The descriptor must contain only objects. Having functions causes errors. It is not checked whether the objects contain functions.

```
[def] = {
   $name: string,
   InstanceOf: function(obj){},
   New: function(desc){},
}
```

- The *$name* is the same as what was used to create.
- The *InstanceOf* takes an object and returns true or false based on whether the object is an instance of the definition. A null or undefined object returns false.
- The *New* optionally takes an argument. If the argument is valid, it must be a subset of the descriptor that was used while defining. otherwise JEEP aborts.

```
[inst] = [def].New()
[inst] = {
   [desc],
   $name: string,
   Clone: function(){},
   Change: function(desc){},
   Equal: function(other){}
}
```

- The *Clone* returns the deep clone of the instance.
- The *Change* is used to change the values in a controlled way. The descriptor given must be a subset of what was used while defining, otherwise JEEP aborts.
- The *Equal* returns true or false based on whether the given object has the same values. Checking the type is the first step of testing equality. So a false could mean either different type or different values.

A.6. Structure

```
[def] = Generator(name, desc)

desc = {
   CONSTRUCTOR: function(){},
   Variables: {},
   Functions: {}
}
```

- The *Variables* must contain only objects. Having functions causes errors. It is not checked whether the objects contain functions.
- The *Functions* must contain only functions; having objects causes errors.

```
[def]  is same as that of record plus some additions.
[def] = {
   $name: string
```

```
   InstanceOf: function(obj){}
   New: function(cmd, desc)(),
   InitNew: function(desc){}
}
```

- The *New* differently from record's. The cmd is multityped. If it is *"init",* JEEP expects the desc to contain a subset of the *Variables* used while defining. Otherwise JEEP aborts. If cmd is a string and not "init", or not string, it is forwarded to the constructor.
- The *InitNew* is an explicit variation of calling *New* with *"init"* and desc.

```
[inst] = [def].New(x), where 'x' means anything
[inst] = [def].New("init", desc)
[inst] = [def].InitNew(desc)

[inst]  is same as that of record except plus some additions.
[inst] = {
   [desc],
   $def: [def]
   $name: string,
   Clone: function(){},
   Change: function(desc){},
   Equal: function(other){}
}
```

The $*def* is same as the definition itself.

A.7. Class

```
[def] = Generator(name, desc)

desc = {
   Flags: string,
   BaseClass: string,
   CrBaseClass: [],
   CONSTRUCTOR: function(){},
   DESTRUCTOR: function(){},
   Variables: {},
   Functions: {},
   Private: {},
   Static: {},
}
```

- The *Flags* property controls the code generation and other aspects.
  - *need-ancestor-access*
  - *manual-base-construction*
  - *using-private-members*
  - *internal-variable-change*
  - *replace-virtual-functions*
  - *trap-invalid-virtual-call*
- The *BaseClass* is a comma separated list of class names
- The *CrBaseClass* is an array of class definitions.
- The *Variables* property must have objects only; having functions causes errors. The variables can have these directives.
  - *set*: creates a setter function for 'var' of the form SetVar: function(v)
  - *get*: creates a setter function for 'var' of the form GetVar: function()
- The *Functions* property must have functions only; having anything else causes error. The functions can have these directives.
  - *const*: renders the function constant
  - *argnum*: validates the argument count passed to the function
  - *argconst*: renders the arguments passed constant
  - *virtual*: renders the function virtual

- *replace*: directs JEEP to replace the base class virtual function with this; can be used only with *virtual*.
- *abstract*: renders the function abstract
- *scoped*: causes the function to create a scope inside which ScopedCreate can be called
- *usepriv*: instructs JEEP that the function needs access to private members
- Apart from functions, only a strings in specific format is allowed in this object. A dollar prefixed string acts as the type descriptor for the arguments of the function with the same name. Ex. *$Print: "Number"* directs that the function *Print* can have only one argument of a *Number* type. The type can be one of these
  - *Number*: allows only numeric arguments
  - *String*: allows only string arguments
  - *Array*: allows only array arguments
  - *Object*: allows only object arguments
  - *record.Record*: allows only Record which must be a record
  - *struct.Struct*: allows only Struct which must be a structure
  - *class.Class*: allows only Class which must be a class
  - *record.Namespace.Record*: only Record which must exist in Namespace (applies to class and structure accordingly)

- The *Private* property can contain both functions and objects. Only function types and directives are allowed but not variable directives.
- The *Static* property can contain both functions and objects. Only function types and directives are allowed but not variable directives.

```
[def]  is a subset of structure and has many items removed and some additions.
[def] = {
   [desc],
   $name: string,
   InstanceOf: function(obj){},
   ScopedCreate: function(args){},
}
```

The *ScopedCreate* is available only if the class has a destructor defined. It can be used only inside a scope. It creates an instance and forwards the arguments to the constructor.

```
[inst] = new [def]
[inst] = new [def](other)
[inst] = new [def]("init", desc)
[inst]= [def].ScopedCreate()
[inst] = {
   [desc],
   $name: string,
   $def: [def],
   $base: {},
   ExternalCall: function(func, args){},
}
```

- While instantiation, if there is exactly one argument and it is same type as the definition, copy construction is triggered and the constructor call is skipped.
- The $*base* exists only if there are overridden virtual functions or *manual-base-construction* flag is used. Just having base classes doesn't create this property.
- The ExternalCall function is available only if the *internal-variable-change* flag is used.

Appendix B

Some Notes for C++ Developers

Only structure and class related notes are listed here as they are the most relevant ones. Even with that, only those that are significant enough are mentioned. The numbering is only for convenience and has not other connotations.

B.1. Similarities

1. classes have constructors and destructors and are invoked in the C++ order
2. copy construction is available in the expected form
3. destructor is called when objects go out of scope
4. classes can have public, protected and private members
5. private members are inaccessible to derived classes as well as external code
6. classes can have static members
7. classes can have constant functions
8. classes can have one or more base classes
9. classes can have virtual and abstract (pure virtual) functions
10. classes with unimplemented abstract functions are not instantiated
11. virtual functions called from constructor and destructor behave as in C++

B.2. Differences

1. structures and classes cannot stand in for each other
2. structures cannot have access specifiers
3. structures cannot have base classes
4. declare define split is only for readability, not resolving dependencies between classes
5. a class definition is generated only after define returns; using declare is not enough
6. init construction is available for structures and classes
7. there are no overloaded functions
8. class constructors can return false to indicate failure or construction
9. upon failed construction partial destruction is always attempted
10. protected and private members reside in a separate sub objects of the instance
11. protectedness can be revoked without changing the code
12. duplicate member names are not allowed in class hierarchy
13. ambiguity resolution in hierarchy is only via wrapper classes
14. many inheritance patterns that are valid in C++ are invalid §7.3.1.
15. access to ancestors (bases of bases) must be explicitly requested via flags
16. base classes are always virtual
17. base class construction order can be overridden
18. virtual call from constructor and destructor can be treated as errors
19. base classes are merged into derived classes and lose their identity