JEEP

A JavaScript Framework for Robust Software Engineering
influenced by Objected Oriented Programming Principles

Design and Development
by
Vinay.M.S

August 2018

Contents

Preface

Introduction

JEEP is an ambitious framework intended to impart features to JavaScript that enables robust software engineering. It is important to establish upfront what I mean by these terms because everyone in our industry give their own.

- *Software Engineering*: Creating reusable, customizable and extensible components of complex structure and behavior.

- *Robustness*: The aspect that makes the structure and semantics meaningful by providing a certain level of guarantee through which we can reason about the correctness of the code and the application written with it. Robustness is not same as defensive programming, but something that inherently imparts such a mechanism so that the programmer can focus on application logic, thereby uncluttering both the code and the mind.

Most of the features Jeep provides are inspired by objected oriented programming (OOP) principles because it is my belief, from my limited experience, that software engineering as I describe is best undertaken with this paradigm. You might argue against the 'object orientation' by citing examples from functional programming etc, but closer inspection of these other techniques reveal a rather ad hoc and imperfect version of what can be only called object orientation. JavaScript being a curious mix of different programming paradigms does offer one such ad hoc and imperfect object orientation, but Jeep makes it much more explicit, structured, formal and extensive. Since I am basically a C++ developer, it amounts to Jeep being a C++ inspired JavaScript framework.

To be clear, Jeep is a JavaScript framework that looks, feels and behaves like C++ for most part.

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 their 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. I would like to argue that one's experience with fragile systems, in general, over a period of time, 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 $\rightarrow$ j seep $\rightarrow$ jeep). As stated, the aim is to elevate JavaScript from a language that does DOM manipulation using scattered functions, to a language suitable for 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 like all improvements to JavaScript happens only in the form of faster engines, even the latest standard updates are just reluctantly added syntactic sugars. Beyond

this, its always yet another megacorp promoting yet another half baked one page application framework with "active community" that is the 23rd clone of a preexisting framework with small variations. Nothing seems to happen at conceptual or structural level. Jeep tries to fill this gap.

A small general purpose testing framework is bundled with Jeep using which it has been tested well. There are nearly 375 tests covering failing and passing scenarios, but 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 beyond saying that Jeep is well tested.

Among the mainstream 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 as I don't use it). It demanded very high level of programmer competence to use it but offered mighty rewards in return. Then came C++. While its explicitly stated goal was to allow better organization of programs, since the problem of computation was already handled rather well by C, it can be interpreted as bringing some order to the unruliness of C. This interpretation proves correct when you read its design motivations. I consider Jeep to be something like that. If not C++ itself, it 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. All this is coincidence, of course, but it's a striking one.

A brief history

I ventured into web development rather late in my career; I was a desktop developer. In December 2012 I quit my last job in order to pursue some product ideas (before the world ended) and in March 2015 I realized that some of them were better packaged as web products, mainly to avoid the hassle of platform specific builds and updates. My initial excitement at the new found freedom a language like JavaScript offers over a language like C++ quickly waned when I saw that I just had a bunch of functions and closures to hack with.

Coming from a more robust and object oriented language, I felt rather claustrophobic in this restricted space. Firstly, there was no semantic validation – a function taking any number of arguments irrespective of its signature is one example. The concept of function signature itsels seem non existent. This may seem flexible but it hinders robustness. Secondly, there is no mechanism for building class hierarchies - there isn't even a concept of a class to be exact – and what exists is a standards sanctioned hack. The lack of virtual function mechanism and multiple inheritance was the primary inconvenient factor.

I researched for frameworks that made these things possible but none 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. Jeep is basically the result of my trying to adapt JavaScript to make me feel at home. Jeep is still a hack, since that's the only way to get these features to work in JavaScript, but it abstracts the hacks through a consistent and well structured interface. This spares applications of repeatedly cobbling together ad hoc and imperfect hacks.

Jeep started out as a modest function and was immediately incorporated into my code base. When I revisited web development many months later, after having been diverted elsewhere, in August 2017, I found Jeep to be severely lacking and decided to improve it. What started out as a five day job ended up taking fifty days. In these fifty days and fifty nights 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 even uploaded it to github hoping to promote it, but I again got diverted elsewhere.

As it happens, when I revisited web development in August 2018, the supposedly new and improved version was found to be not up to the mark, so I decided to improve it again. Many ideas seemed fragile and in need of more thought and polish. The old implementation being basically a cumulative hack job was unreadable leave alone conducive to updates, so I ended up rewriting it from scratch. It was sort of reimplementing a specification, although several concepts were improved or discarded in the process. The documentation also needed revamping both due to revamping of the framework, and the inherent imperfections in exposition. This time it took fifteen days straight to do all the same things all over again.

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.

About the document

Due to its goals, Jeep is a complex framework although it is quite simple to use. Such a framework cannot be introduced effectively by just listing out the features and API's with some examples. Further, Jeep is a conceptual framework, in that it tries to import alien concepts to the language. Therefore a document of this size and nature was necessary.

The document discusses every aspect of Jeep thoroughly. After the **Introduction** chapter, each chapter is dedicated to one or more features as the titles indicate. Apart from these, the **Example** chapter demonstrates how Jeep can be used in real world application with a small GUI library, and a GUI tool using thee library, and the **Internals** chapter discusses the implementation details, performance and testing briefly. The document ends with the **Reference** chapter which, as the names suggests, is purely a technical documentation, meaning it is dry and lists things in a matter of fact way. There are also few demonstration code snippets. The chapter is intended to be a quick reference without having to deal with exposition.

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, although I mention and breeze past them to help with the narration.

The intended readers are of two groups – C++ developers who have landed in JavaScript land like me, and JavaScript natives who want to explore more ideas and frameworks. Due to this, I often compare and contrast Jeep with C++. 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 very briefly to give native JavaScript developers some idea about what it is all about. The C++ veterans are expected to ignore these bits.

As is expected from a document of this nature, there is generous amount of code, which adds to the length of the document. All code is available in an accompanying file.

Conclusion

I fully expect a mixed response to this framework and very slow adoption. A lot of things Jeep 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 programmers are, but I hope most of those heads fall backwards and ponder a few moments later.

I don't think my being an unknown engineer with no public credentials must hider appreciation and adoption of Jeep since the code, concepts and everything else is there for you to consider at your own pace. This was another motivation to write this rather large document - I wanted to discuss things rather than simply mention them.

I assume and hope that Jeep will appeal to and be appreciated by C++ veterans who have taken up JavaScript and open minded native JavaScript developers equally.

Jeep has been central to my JavaScript programming since its inception and has greatly benefited me. I hope you find it useful too.

About Me

I am primarily a C++ developer with almost 14 years experience but I am not an expert in the language, or in anything for that matter. I have not amassed six digit points on online forums. I cannot write code to balance a binary tree on a whiteboard, or anywhere. I am more of an idea driven engineer. I call myself an *iota to alpha developer* – take an iota of an idea and create an alpha version of the product. I constantly find myself having to learn new things or improve my knowledge in the course of development, which I find thrilling. For most of my career I have been a lead developer and team leader working on small iotas. Though I am a generalist I have somehow always ended up doing GUI for scientific applications.

I had one such iota that made me quit my job in December 2012. The unrestricted freedom caused the iota to branch out into a dozen of them and allowed me to spend extended amount of time working on each one of them iteratively. Many were discarded, many were added, many were improved. As of writing this document I am still unemployed, trying to take some of them to alpha and a few of the alphas to beta and hoping to launch a company.

I am from Bengaluru, India. Apart from writing software, I have a lot of interests in diverse fields, but since writing software can bring me the financial backing necessary to be able to pursue others, now and later, I am a professional software engineer. I strive to be self sufficient and I thrive working alone. I would describe my personality in two words – ambitious (within ethical limits) and autodidactic. My life goal is betterment of society via my company, products and profits.

I want to publicly acknowledge the infinite support and patience my parents showed me in this extended period of joblessness, and encouragement in general. Without them I might have had an insipid career where I aimed to become yet another VP in a megacorp instead of having the conviction to dream and pursue a more meaningful career, life and goals.

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

---

* Before the third iteration in 2018, I did a brief search again and landed on something called ease.js which is part of GNU. I was pleasantly surprised to see something very close to Jeep already exist. Interestingly, many implementation details to get some features to wotk and the problems they begat were nearly identical to what Jeep2017 had, which are solved conclusively in Jeep2018.

1. Introduction

1.1. Preface

What is known as object oriented programming (OOP) is an amalgamation of different, and arguably independent, concepts. Therefore, I want to start off by briefly discussing what I mean by OOP and how Jeep allows it.

If OOP is seen as an organizational technique where related data and functions are grouped together as a unit, and each unit gets its own copy of data, then OOP is more widespread than it appears. Languages like C++ make it more explicit with `class`, languages like C allow an inferior form of this to be handmade by the programmer working with `struct`, while in functional languages like Scheme it is not evident unless you reason about how the programs work and understand closures. JavaScript being a curious mix of different programming styles, and being a prototype based language, does allow OOP at a level that is in between C and C++. However, the way to achieve this is more inferior and crude than what C programmers have to do. The latest features added to the language such as `class` tries to ameliorate this, and is probably sufficient for simple OOP style, but Jeep offers a lot more and offers them at C++ level.

If OOP is seen as a technique to achieve dynamic behavioral abstraction where an agent that provides the requisite behavior is not known until runtime when the behavior is requested for, then, again, OOP is more widespread than it appears. Languages like C++ make it more explicit with virtual functions, languages like C allow an inferior form of this to be handmade by the programmer using function pointers, while in functional languages like Scheme it is much simpler due to functions being first class objects and conducive to being passed as arguments to other functions. JavaScript being a curious mix of different programming styles, and being a prototype based language, does allow OOP in both C++ way and Scheme way, but it is inferior to C++ and as inflexible as Scheme. It is inferior and inflexible because the intended behavior cannot be dynamically resolved automatically and needs programmer logic to decide which behavior must be invoked based on the context. Jeep offers OOP at C++ level, and even improves on some of the drawbacks that C++ has in this regard.

If OOP is seen as a technique to incrementally build objects by extending existing objects and inheriting their behavior, then OOP is limited to languages like C++ which offer it explicitly. Other languages have to go through hoops to get this working. JavaScript being a prototype based language does allow OOP but it is quite limited. Actually, OOP exists simply as a consequence of JavaScript objects really being just one big key-value map. Further, what is the fun in extending only one object? Jeep offers OOP at C++ level, and even improves on some of the drawbacks that C++ has in this regard.

Jeep doesn't try to import all of C++ into JavaScript – that's futile and foolish – but imports all those concepts and behaviors that fortify and improve JavaScript, thereby enabling robust software engineering with the language. As noted, Jeep improves on several things that C++ does wrongly in my opinion. I took this opportunity to implement them in a way that I feel they ought to have been. It might appear quite amusing that an unknown engineer with no public credentials has such strong opinions that challenge the collective intelligence of a whole committee of accomplished computer scientists and engineers that designed and improved C++. I concur.

1.2. Overview

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 is written in standard JavaScript (ECMAScript 5), so it should work on all standards complaint browsers, probably even on Internet Explorer.

Jeep has the following qualitative features.

- promotes writing well organized and good looking code
- makes the code intuitive and readable
- makes the code easily extensible and maintainable
- provides semantics to write robust code
- encourages productivity by removing boilerplates
- offers a range of objects and flexible features to model data and behavior effectively

Jeep has explicit notion of objects. For some objects, you first define them and then instantiate them. There are three such objects of increasing complexity: `record`, `structure` and `class`. A record is intended to be pure data, a collection of variables packed as a unit. A structure is intended to provide simple functionalities such as data structures and other utilities. A class is the most complex and feature rich object, and is intended to model the problems (or solutions) that have complex structure and behavior.

As mentioned, you have to first define the objects. The step is called generating the definition. There are two ways to generate definitions: creating and registering. With creating, the generating function returns the definition as the result using which you can instantiate the objects. These definitions are essentially local to the scope where the generating function were called, so they must be sent out of the scope if they need to be used elsewhere. One way of doing this is having them as global objects. but such objects are vulnerable to accidental reuse, especially when Jeep definitions are used in third party code. Of course, when this happens the script fails in no uncertain terms and you can easily know about it, but Jeep helps you avoid the hassle in the first place with registering mechanism.

When definitions are registered, they are stored in a common database, which is deep inside the framework, so nothing short of deliberate mischief can corrupt them. The definitions are to be retrieved before using. The retrieved definitions are less likely to be accidentally reused since they are going to be used in a known context. Registering has another important advantage over creating them locally. The definitions are accessible throughout the application since the database is the global scope, albeit hidden away safely. This aspect allows sharing of the definitions easily. By extension, it allows referencing a definition not in the current scope. This referencing part particularly helps in implementing one of the important Jeep robustness features.

Apart from the three objects mentioned above, there are two more objects that are a bit different: `fields` and `groups` (no, there are no rings). These objects don't have the define-instantiate way of working. They are sort of pre instantiated and can be used directly. A field is the exact opposite of a record, in that it is pure functionality with no data. It is essentially a package of related functions. It is intended to represent simple API. A group is a feature rich field that can have data. It is intended to represent API that have initial settings or need to be initialized dynamically.

Beyond these five, there are two related objects - `namespace` and `library` - that are more organizational in nature. A namespace is a name disambiguating mechanism. As mentioned above, registering definition uses a common database, and it is to be expected that names clash when code from different sources are used. A namespace helps to deal with this in a transparent way. A library is a higher level object that helps logical grouping of objects. It is really a namespace underneath, but it offers enough abstraction and different behavior that it can be considered a different object.

All these objects work in the context of an environment, which is also an object but more of foundational and not operational or organizational.

Robustness features are plenty, and are applicable to both variables and functions. Most of these features are quite alien to JavaScript, and might need a bit of reflection before they can be appreciated. Upon reflecting, it will become evident that almost all the features are implemented in an ad hoc and imperfect form in existing JavaScript code; some of the features are too complex to implement ad hoc, so they are not likely to be seen in the wild. What Jeep offers is a more comprehensive solution.

These are the available robustness features.

- access restriction to members
- enforcing variable constantness
- enforcing function constantness
- enforcing function argument count during invocation
- enforcing function argument types during invocation
- enforcing function argument constantness
- providing guaranteed mechanism to release resources
- enforcing sound polymorphism

Beyond all these, Jeep also offers some utilities, basically functions and structures, for applications' benefit, but its up to the applications to use them. They are in no way connected to the rest of the framework except that they are used in Jeep's own implementation. So they are put to use and tested in a real world scenario and applications could use them with good amount of trust.

1.3. Example

Although Jeep is feature rich, the main attraction is classes with virtual functions and multiple inheritance. It is also the primary motivating factor to creating the framework in the first place. So the example showcases them.

Consider this code for a while. Every part is briefly explained later, and more comprehensively in the relevant chapters.. I suspect C++ developers might find this amusingly cute, and JavaScript natives might think it is weird and verbose.

```
JEEP.InitFramework();
let DemoEnv = JEEP.CreateEnvironment({client: "jeep-aware", mode: "development-mode"});

DemoEnv.RegisterClassDef("Animal", {
	CONSTRUCTOR: function(n){
		this.name = n;
	},
	PUBLIC: {
		liveOneMoment: function(){
			this.breathe();
			this.move();
		}
	},
	PROTECTED: {
		breathe__abstract: function(){},
		move__abstract: function(){},
	},
	PRIVATE: {
		name__get: "",
	}
})
DemoEnv.RegisterClassDef("Fish", {
	EXTENDS: ["Animal"],
	PROTECTED: {
		breathe__virtual: function(){
			cout(this.$name, this.getName(),"breathing hrough gills")
		},
		move__virtual: function(){
			cout(this.$name, this.getName(),"moving using tail fins")
		},
	}
})
let Human = DemoEnv.CreateClassDef("Human", {
	EXTENDS: ["Animal"],
	PROTECTED: {
		breathe__virtual: function(){
			cout(this.$name, this.getName(),"breathing through nose")
		},
		move__virtual: function(){
			cout(this.$name, this.getName(),"moving by walking")
		},
	}
})
DemoEnv.RegisterClassDef("Mermaid", {
	EXTENDS: ["Fish", Human],
	PROTECTED: {
		breathe__virtual: function(){this.$base.Human.breathe()},
		move__virtual: function(){this.$base.Fish.move()},
	}
})
let Mermaid = JEEP.GetObjectDef("Mermaid");
let m = Mermaid.New("Mermy");
m.liveOneMoment();
```

```
Mermaid Mermy breathing through nose
Mermaid Mermy moving using tail fins
```

1.3.1. The Basics

The framework is packed inside the `JEEP` object, referred to as the main object.

```
JEEP = {
       InitFramework: function(){},
       SetStdErr: function(logger){},
       CreateEnvironment: function(info){},
       ObjectDefExists: function(name){},
       GetObjectDef: function(name){},
       Typedef: function(newtype, existing){},
       RegisterLibrary: function(name, initFunc){},
       Equal: function(a, b){},
       ValEqual: function(a, b){},
       Utils: {},
       impl: {},
}
```

The `impl` object contains all the implementation details and is not supposed to be touched. The `Utils` object contains utilities mentioned earlier. The rest of the functions will be encountered while demonstrating different objects in their respective chapters, so the explanation will be deferred till then.

The framework must be initiated before it can be used.

After that, before doing anything useful, you have to create an environment, which is composed of three properties: `mode`, `client` and `flags`. The first two must always be given explicitly, to the creating function, otherwise Jeep issues an error and aborts.

The `mode` property impacts the behavior of the framework significantly. It takes two possible values: `development-mode` and `production-mode`.

In development mode, all validations are always done which are of three kinds: syntax, semantics and runtime. In production mode, syntax and semantic validations are completely skipped. Some runtime validations can be selectively retained by using flags in object declaration, which is the second argument given to the generating functions. These flags are object specific and will be discussed in the respective chapters. The general usage pattern with Jeep is to first use the development mode to avail all the robustness benefits and then switch to production mode to get maximum performance.

The `client` property doesn't cause any behavioral changes by itself, rather it is intended to be a clue to the code that provides definitions such as libraries, to use run time flags in order to cause the behavioral changes. It takes two possible values: `jeep-aware` and `jeep-agnostic`.

The jeep aware clients can be expected to be have been themselves developed in two modes, thereby being fully validated. Such clients can be trusted to do the right semantic thing and all robustness features can be turned off to get maximum performance benefits in production mode. In contrast for jeep agnostic clients, it can be expected that they treat Jeep object instances as any other JavaScript object and thoroughly abuse them, and some robustness features might be retained at the cost of a small performance hit. Having jeep agnostic clients is not that hard to imagine - a library could use Jeep but return only the instantiations rather than the definitions, which the clients of the library use as ordinary JavaScript objects.

The `flags` property is a string, and as mentioned it is optional. These flags affects the validations undertaken, but are applicable only to specific objects, and shall be explained along with the relevant objects.

The created environment object looks like this.

```
Environment = {
	IsDevMode: function(){},
	IsClientJeepAware: function(){},
	CreateRecordDef: function(name, def){},
	RegisterRecordDef: function(name, def){},
	CreateStructDef: function(name, def){},
	RegisterStructDef: function(name, def){},
	CreateClassDef: function(name, def){},
	RegisterClassDef: function(name, def){},
	CreateField: function(where, def){},
	RegisterField: function(name, def){},
	CreateGroup: function(where, name, def){},
	RegisterGroup: function(name, def){},
	CreateNamespace: function(){},
	GetLibrary: function(name){},
}
```

The functions `IsDevMode` and `IsClientJeepAware` are simple query functions that return a Boolean. They are intended to be used by the clients and applications as described earlier. The rest of the functions will be encountered while demonstrating different objects in their respective chapters, so the explanation will be deferred till then.

1.3.2. Definition and Instantiation

As mentioned in 1.2, there are two ways of generating the definition. The `CreateClassDef` returns the definition while the `RegisterClassDef` adds the generated definition to a common database. Both are present in the environment object, and as you can see, this pair exists for record and structure as well; field and groups are a bit different so they have a different function signature.

The functions takes two arguments. The first is the name of the object, and the second is the declaration which is processed to generate the definition. All the property names in the declaration are in uppercase in order to draw attention to them. More about all this will be discussed in later chapters.

Registered objects are accessed via the `GetObjectDef` present in the main object. The function takes one argument, which is supposed to be the same name used while registering.

The generating functions are present in the environment object but the access function is in the main object because the generation phase is affected by the mode of the environment, which controls validation, whereas accessing simply looks up the database, which is straightforward both in operation and error handling.

The definitions of records, structures and classes automatically get three functions: `New`, `InitNew` and `InstanceOf`. The first two are used to instantiate, while the third is used to check if an object is an instance of the definition.

As with generating definitions, the instantiation part remains consistent across the three objects as well. There do exist slight variations, and other methods to instantiate some of them, and they will be discussed in their respective chapters.

Instances are sealed so that new properties cannot be accidentally added, or existing ones be deleted deliberately. This is related to getting good performance, and will be explained in 1.3.6.

1.3.3. Member Variables and Functions

In relation to Jeep objects, both definitions and instances, properties are called members. Class members have multiple access privileges, which is based on the sub objects inside which they are defined. The constructor is a special function, and the first one that is automatically invoked upon instantiation. It is meant to do all the necessary instantiation.

The values declared with the variables are taken to be their default values, to which the variables are set when instantiated with the `New` function. While variables values can always be changed, they can be initiated to non default values in a much simpler way, in one function call, by using the `InitNew` function to instantiate.

There are a few member variables automatically added to both instances, which help them self identify and do things reflexively, and hence called reflective variables. They are intended to relieve the instances of having to rely on external objects that must be captured as closures and are vulnerable to accidental abuse. They are prefixed with a dollar character in order to stand out. For this reason, no members can be declared with a dollar prefixed name.

The variable `$name` has the same value as was used while generating the definition. This essentially identifies the name of the type. It exists in the definition also, and is the only one added there. This is useful in creating new types with the `Typedef` function. The variable `$def` is exactly same as the generated definition. This helps member functions to instantiate more of their own kind. or test the types. There are more such variables that will need a context to be appreciated, and hence will be deferred for later.

1.3.4. Directives

Another consistent thing across different objects is the usage of member directives. Directives are basically Jeep specific keywords that impart special behavior or instruct the framework on definition generation. A full fledged programming language would use space separated prefix or suffix along with member names, but since Jeep is basically JavaScript, and cannot invent syntax out of thin air (or thick for that matter), Jeep improvises. It uses decorated names.

The general pattern is this: The member name is suffixed with two underscores to separate it from the directives. There can be more than one directive, and they are to be separated by a single underscore. The underscore count is fixed for simplicity. The double separation was enforced to make the name more prominent. The directives are suffixed in contrast to C++ and most languages in order to draw attention to the member name, otherwise you have to scan the line to the right just to know the member name, which is not good in my opinion. The directives essentially qualify the members. Due to the syntax, Jeep is more Arabic than English.

This is how I had environed directives from the start, though Jeep2017 had the order reversed with a slightly ugly syntax compared to Jeep2018, until I saw what ease.js did. For a moment I felt like I had over complicated things but I quickly realized it wasn't so. Using something like

```
"breathe virtual": function(){}
```

might seem easy to write than

```
breathe__virtual: function(){}
```

but it poses three issues. Firstly, you have to do extra string processing to weed out invalid strings, whereas with Jeep way you simply piggyback on JavaScript syntax. Secondly, it is much tiring and cumbersome to use quoted property names instead of directly typing them out. Thirdly, whenever you decide you need a directive while refactoring, you have to convert it to a quoted name, which is quite tiring; and if you always use quoted name in anticipation of adding a directive, it takes us to second issue.

1.3.5. Transmissibility

Features imparted by some directives are transmissible. What this means is that all functions that are part of the call chain started by the function with such directives are imparted the effect of the directives even if the functions are not defined with them. Thus, non member functions invoked will also be subject to this behavior. This is a side effect and not a designed feature.

This happens mainly due to the nature of JavaScript. Jeep has to work with whatever the language offers, and there is nothing in the language that allows to intercept function calls which might have helped by blocking this side effect.

Transmissibility is desirable in some cases but not at all in other, so it can't be judged too harshly. Even if JavaScript had provide the intercepting feature, transmissibility might have had to be implemented using clever hacks.

1.3.6. Performance Benefit

Using Jeep objects improve the performance of the code over using plain JavaScript objects. This benefit applies to records,, structures and classes for the reasons that will be explained, but it is more relevant to records since they act as data building blocks in an application and are likely to have hundreds of instances.

A bit of engine internals is needed to appreciate how performance benefits with records. What I summarize here is based on Chrome engine internals as explained by Franziska Hinklemann, an engineer working on the engine. Watch her talks on YouTube for more in-depth insight.

As you are already probably aware, modern JavaScript engines don't interpret the code, but compile it into native code using JIT compilers. The engines do a lot of optimizations in order to get maximum performance by collecting lots of execution statistics. One such information is the shape of the object, which is basically the number of properties an object has and what names they have. Engines treat objects with different shapes as different types and generate code accordingly. The shape of the object is affected when properties are added and removed. When

this happens, the internal type information has to be changed, which leads to the engines to reconsidering optimization done for the object in question, and going back to the de optimized native code.

A typical JavaScript code usually engages in setting property values; rarely do they do the deletion. In the course of doing this, if a property is misspelled, the code ends up adding a new property, and causing the engine to de optimize.

Records mitigate the situation by providing `InitNew`. Since this is a validated function, you cannot accidentally change the object shape during instantiation. Then, since the instance are sealed, there is no way to add and remove members on the instance. Records tackle the problem in both scenarios and provide a massive boost to performance, which will of course only be noticeable when hundreds of instances are used such that their member values are modified.

1.4. Nature of Jeep

Although Jeep is often referred to as a framework, close observation will show that it is much more. It is also a programming language, a compiler and a development environment.

- 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 intuitively.

- It qualifies as a programming language because it has syntax, semantics, and imposes many rules to keep things consistent and logical.

- It qualifies as a compiler because, essentially, Jeep code is compiled to plain JavaScript code. During compilation, 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 development mode, where all validations are always effective, is tuned to offer the developers maximum debugging assistance. The production mode, where most validations are skipped unless explicitly retained, is tuned to give the users maximum performance.

There seem to be not much effort put to discipline the rich brat that is JavaScript that has grown up without adult supervision beyond some standards update which are not effective. Had they been, I wouldn't have created Jeep.

One popular solution is to write code in some other scripting language and then compile it into JavaScript. This other 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 since 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 compiler written in JavaScript to compile JavaScript code and generate 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 debug 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. The rules pertain to both syntax and semantics. For example, using a directive like overridden is a syntax error because such a directive doesn't exist in Jeep, whereas using virtual directive on a field function is semantic error because fields don't support polymorphism. 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. Due to the nature of JavaScript, both compile time and runtime errors always abort the script.

2. Record and Structure

Records and structures are very similar in syntax and behavior, so they will be explained together Only object specific features are discussed here; for features that are common to multiple objects, refer to 1.2 and 1.3.

2.1. Record

A record is just a named collection of data. It is Jeep's replacement for plain objects. Records are intended to be used as entries for arrays and values in maps, and, essentially, become the building block of applications. For a language like JavaScript records not only give the much needed structure, but also help improve performance.

2.1.1. The Basics

Consider this code for some kind of file processing utility.

```
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 look at it objectively, 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 repetition, particularly, is against the core principle of software engineering and is quite undesirable. Records are meant to solve these problems.

Now consider the same code with records.

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

DemoEnv.RegisterRecordDef("FTI", {
    src: "",
    dest: "[home]",
    flags: Flags.F_COPY,
})

function event1(){
    //...
```

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

The values assigned to the variables in the declaration are considered default values. The members will be set to these when records are instantiated using the `New` function. This function does not take any arguments; if given it generates runtime error.

To instantiate with non default values, the `InitNew` function is to be used. The function takes exactly one argument which is supposed to be an object containing properties that are a subset of the variables mentioned in the record definition. If any of these rules are violated, it generates runtime error. The ones that are not provided in the input object take on the default values.

I hope you can see how efficient and productive it is with records compared to plain JavaScript code 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. 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 are 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.

2.1.2. Extension

We often see that some data that we used seem a superset of other data. For instance, a 3D point and a 2D point. In these cases its natural to use composition which looks something like this.

```
let pt2d = DemoEnv.CreateRecordDef("2dPoint", {x: 0, y: 0})
let pt3d = DemoEnv.CreateRecordDef("3dPoint", {
       2dp: pt2d.New(),
       z: 0,
})
```

However, its evident from looking at the code that the usage will be extremely cumbersome. Jeep offers a better solution which is more productive.

Records can extend other records. The record being extended is called base record and the one extending is called the derived record.

```
let pt2d = DemoEnv.CreateRecordDef("2dPoint", {x: 0, y: 0})
let pt3d = DemoEnv.CreateRecordDef("3dPoint", {
       EXTENDS: [pt2d],
       z: 0,
})
let p = pt3d.InitNew({y: -1, z: 10})
cout("3d point",p.x,p.y,p.z)
```

```
3d point 0 -1 10
```

The `EXTENDS` property of the declaration is an array that contains either names of registered records or record definitions directly.

With extension, not only the members can be accessed directly as if they were declared in the derived record, the derived records can be instantiated with `InitNew` directly. These two points are related of course, but the effect they have on usability is different.

2.1.3. Abuse

Although records are meant to be pure data with only variables and no functions, due to functions being the so called first class objects in JavaScript, record members can be functions too. The consequence of this is that you can hack a structure out of records. This is due the nature of JavaScript where a container of a function automatically becomes the `this` object. Doing something like this is possible but not desirable.

```
let Person = DemoEnv.CreateRecordDef("Person", {
       name: "",
       show: function(){cout("name:", this.name)}
})
let p = Person.InitNew({name: "unknown"})
p.show();
```

```
name: unknown
```

This is considered abusing records. The important thing here is that, even if a function is used as a variable, it is considered pure data and not put through special processing like functions in other objects are. So use functions only as a data, may be to store a callback, but not to perform actions. If you need action, choose other objects.

2.2. Structure

A structure is in between a record and a class in terms of utility and features. The syntax is similar to record, but there are also several differences. The main one is that structures can have functions, and they are treated as functions and put to special processing. Secondly, structures cannot extend other structures. It is designed so to keep things simple and focused. If you need extension, use classes. Structures are meant to implement utilities such as data structures, which don't lend themselves to be extended but are made to be used in composition.

2.2.1. The Basics

Consider this example. It implements a rudimentary vector (the maths one not the medical one).

```
let Vector = DemoEnv.CreateStructDef("Vector", {
	CONSTRUCTOR: function(rawArr){
		this.arr = Array.from(rawArr);
	},
	arr__private: [],
	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 vec = Vector.New([1,2,3,4,5]);
let res = vec.ApplyOperator('*', vec.ApplyOperator('*'));
res.Print();
try{vec.ApplyOperator("*",10)}catch(e){cout(e.message||e)}
```

```
Vector: 1,8,27,64,125
Vector.ApplyOperator expects a Vector
```

The `CONSTRUCTOR` is a special function which is called first automatically upon instantiation if one exists. This is supposed initialize the instance. Both `New` and `InitNew` invoke the constructor. The functions behave identically as the record functions do, but there is a restriction. For structures, the `New` function can take arguments, but only if a constructor is defined. This is enforced in order to make the code meaningful: what is `New` supposed to do with the arguments if it has no constructor to forward it to?

Further, since the `InitNew` function already does value instantiation, a temporary reflective property `$initnew`i s added to the instance and set to true to allow the constructor to decide how to initialize the instance if it still needs to. It is deleted immediately after the constructor returns.

```
let Struct = DemoEnv.CreateStructDef("Test", {
	value: 1,
	CONSTRUCTOR: function(){
		if(!this.$initnew)
			this.value = 33;
		cout(this.$name, "value =", this.value)
	}
});
Struct.InitNew({value: 10});
Struct.New();
```

```
Test value = 10
Test value = 33
```

2.2.2. Access Restrictions

The other major difference a structure has compared to record is that it can have private members. By default all members have public access for simplicity, so private restriction must me explicitly indicated via the `private` directive. This is the only directive valid for structures. Private members are accessible only by member functions or other instances of the same type. Anyone else accessing generates runtime error.

The access restriction is always enforced in development mode, but can be selectively retained in production mode via flags.

```
function test(env, retain){
	let Struct = env.CreateStructDef("Test", {
		PMC: retain ? "pmc-memaccess" : "",
		value__private: 1,
		print: function(){cout(this.$name, "value =", this.value)}
	});
	if(env.IsDevMode()) cout("development-mode");
	else cout("production-mode( access check", retain?"retained":"not retained",")")
	try{cout("private value:",Struct.New().value)}catch(e){cout(e)}
}
test(JEEP.CreateEnvironment({mode:"development-mode"}))
test(JEEP.CreateEnvironment({mode:"production-mode"}))
test(JEEP.CreateEnvironment({mode:"production-mode"}), true)
```

```
development-mode
JEEP: Attempt to read private variable 'Test.value' detected. Aborting...
JEEP aborted
production-mode( access check not retained )
private value: 1
production-mode( access check retained )
JEEP: Attempt to read private variable 'Test.value' detected. Aborting...
JEEP aborted
```

The property `PMC` stands for "production mode checks". It is a comma separated string containing flags. The only flag relevant to structures is the `pmc-memaccess`. Using other flags generates compile time error.

Access restrictions are transmissible, and the effect is explained in 3.1. However, since structures are simpler than classes, transmissibility is left as is. This might be changed in the next version.

3. Class

A class is the most complex and most versatile object that is a sort of superset of features of the other objects. Consequently, there is lot to discuss and this is going to be a long chapter. Although 1.3 gives a glimpse of what a class looks like, and also explains things briefly, much more needs to be discussed. For sake of better organization, classes are defined in two chapters. This chapter talks about classes as objects. The hierarchy related aspects will be discussed in chapter 5 which bears a rather suggestive title. Only object specific features are discussed in these chapters; for features that are common to multiple objects, refer to 1.2 and 1.3.

All robustness features that Jeep offers are actually class specific features, and they are applicable to class like objects such a field and groups, as will be explained in the respective chapter. Due to Jeep having to make do with whatever JavaScript offers as a language, the robustness features have to be divided into two: strong robustness and weak robustness. Everything could have been made strong, of course, but that would lead to either unnatural usage patterns or severely prohibitive performance issues. So Jeep chose the middle ground by offering at least weak robustness but documenting it well enough to bring it to programmer attention. What is meant by strong and weak will be explained after all features have been discussed.

3.1. Access Restriction

Class members can have three kinds of access permissions: public, protected and private. Public members are accessible to all code. Protected members are accessible only to member functions and derived classes. Private members are accessible only to member functions. The members acquire these restrictions based on where they are defined in the declaration. The restriction can be retained in production mode with the `pmc-memaccess` flag in the `PMC` property.

```
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		pubval: 0,
		pubfunc: function(){
			cout("pubval:",this.pubval,"protval:",
				this.protval,"privval:", this.privval)
		},
	},
	PROTECTED: {
		protval: 100,
	},
	PRIVATE: {
		privval: 200,
	}
})
let c = Class.New();
c.pubfunc();
try{c.protval=0}catch(e){cout(e)}
try{c.privval=0}catch(e){cout(e)}
```

```
pubval: 0 protval: 100 privval: 200
JEEP: Attempt to write protected variable 'Class.protval' detected. Aborting...
JEEP aborted
JEEP: Attempt to write private variable 'Class.privval' detected. Aborting...
```

```
JEEP aborted
```

3.1.1. Accidental Friends

There is a concept of friend functions and objects in C++. These friends are non member entities but can access non public members as if they were members of the class. Jeep doesn't provide this feature because the access privilege that exists for member functions can be inadvertently transferred to non member when they are invoked functions by the member functions (due to transmissibility 1.3.5), thus accomplishing the goal already! Consider this code.

```
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		callback: function(f){f(this)}
	},
	PROTECTED: {
		protval: 100,
	},
	PRIVATE: {
		privval: 200,
	}
})
let c = Class.New();
c.callback(function(inst){
	cout( "protval:", inst.protval, "privval:", inst.privval)
});
```

```
protval: 100 privval: 200
```

3.1.2. Explicit Foes

The access privilege can be explicitly revoked by using the `ExternalCall` function to invoke non member functions. This function is automatically added to the instance (not the definition).

```
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		pubval: 0,
		callback: function(f){f(this)},
		callbackExt: function(v, f){this.ExternalCall(f, this, v)}
	},
	PROTECTED: {
		protval: 100,
	},
	PRIVATE: {
		privval: 200,
	}
})
let c = Class.New();
try{ c.callbackExt(100, function(inst, val){inst.protval = val})}catch(e){cout(e)}
```

```
JEEP: Attempt to write protected variable 'Class.protval' detected. Aborting...
JEEP aborted
```

The function takes at least one argument, the function to be invoked. The additional arguments are optional and are simply forwarded t the invoked functions. This function is always added to the instance, even when no restriction exists so as to not break the code.

3.2. Variables

A class must have variables, either its own or inherited from base classes. Not having variables generates compiler error. This is enforced to keep the code meaningful: what is the purpose of having a bunch of functions that are invoked on an instance if they work on outside data?

Class variables can take more directives than structure variables. These directives help in code generation as well as implement some robustness features.

3.2.1 Getter and Setter

Variables are ideally implementation details and not needed to be exposed to outside code. However, there is often a need to expose some of these because variables also act as parameters. Such parameter variables also need to be set outside instantiation. You can of course ask the clients of the class to access the variables directly, but that is not the right approach; you should provide a function for the task. But it's a chore to add these simple functions. Jeep helps you automate the generation of these functions with `get` and `set` directives.

```
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		count: function(){this.counter+=this.step}
	},
	PRIVATE: {
		counter__get: 0,
		step__get_set: 1,
	},
})
let c = Class.New();
c.count();
cout("c.counter", c.get_counter())
c.set_step(2);
c.count();
cout("c.counter", c.get_counter())
```

```
c.counter 1
c.counter 3
```

The `get` and `set` directives generate simple functions: the former simply returns the value and the latter simply assigns the value, nothing more nothing less. The names of the generated functions are simply the directives prefixed to the variable name to make it camel cased. The only restriction is that the `set` directive can't be used with constant variables, for obvious reasons. If you try it you will get compiler error.

3.2.2 Constant Variables

Constant variables are those whose value cannot be changed once the instantiation is finished. They can however be changed during instantiation by the constructor. This is allowed to give

some flexibility to set them at runtime. After this, no one can change it without raising runtime error, not even member functions. The directive `const` renders the variable constant.

```
let Class = DemoEnv.CreateClassDef("Class", {
       PUBLIC: {
                       value__const: 10,
                       obj__const: {val: 0},
                       changeValue: function(x){this.value = x},
                       changeObj: function(x){this.obj.val = x},
                       callback: function(f){f(this)},
                       print:function(){cout("value:",this.value,
                              "obj.val",this.obj.val)}
              }
})
let c = Class.New();
try{c.value=22}catch(e){cout(e)}
try{c.obj.val=22}catch(e){cout(e)}
try{c.changeValue(33)}catch(e){cout(e)}
try{c.changeObj(33)}catch(e){cout(e)}
c.callback(function(inst){
       try{inst.changeValue(999)}catch(e){cout(e)}
       try{inst.changeObj(999)}catch(e){cout(e)}
})
c.print();
```

```
JEEP: Attempt to change constant variable 'Class.value' by non member function
detected. Aborting...
JEEP aborted
JEEP: Attempt to change constant variable 'Class.value' detected. Call trace:
[Class.changeValue]. Aborting...
JEEP aborted
JEEP: Attempt to change constant variables 'Class.obj' detected. Call trace:
[Class.changeObj]. Aborting...
JEEP aborted
JEEP: Attempt to change constant variable 'Class.value' detected. Call trace:
[Class.callback,Class.changeValue]. Aborting...
JEEP aborted
JEEP: Attempt to change constant variables 'Class.obj' detected. Call trace:
[Class.callback,Class.changeObj]. Aborting...
JEEP aborted
value: 10 obj.val 999
```

The error message shows the call trace whenever its available, which is only when member functions are the offenders. There is no way to detect which non member functions attempted to change the value. Even when the trace ends due to member functions invoking non member functions, the partial trace should help with debugging. In production mode, the validation can be retained using `pmc-constant-variable`. Notice that `object.val` is changed! This is because sometimes variable constantness is weak. More on that in 3.7.

3.3. Functions

The special functions, constructor and destructor, will be discussed in 3.4 and 3.5 respectively. Two directives – virtual and abstract – relate to class hierarchy, so they will be discussed in the respective chapter. Functions take a lot more directives and are lot more potent in providing robustness.

3.3.1 The const Directive

This directive renders a function constant. What this means is, such the function can only read the variable values and not change them. While a constant variable makes a variable constant to all functions, a constant function makes all variables constant for that function. In production mode, the validation can be retained using `pmc-constant-function`.

This little beauty of a feature is extremely useful in establishing robustness and semantic soundness. It allows reasoning about the correctness of code by inspection to the first order of approximation since it is guaranteed that constant functions don't cause mutation to the internal data. This way, you can engage in intelligent debugging and not waste time by putting breakpoints all over the place.

```
let Class = DemoEnv.CreateClassDef("Class", {
       PUBLIC: {
              value: 10,
              change__const: function(x){this.value = x},
              callback__const: function(f){f(this)},
              print: function(){cout("value:", this.value)}
       }
})
let c = Class.New();
try{c.change(33)}catch(e){cout(e)}
try{c.callback(function(inst){inst.change(44)})}catch(e){cout(e)}
try{c.callback(function(inst){inst.value=55})}catch(e){cout(e)}
c.print();
```

```
JEEP: Attempt to change the variable 'Class.value' inside constant function detected.
Call trace: [Class.change]. Aborting...
JEEP aborted

JEEP: Attempt to change the variable 'Class.value' inside constant function detected.
Call trace: [Class.callback,Class.change]. Aborting...
JEEP aborted

JEEP: Attempt to change the variable 'Class.value' inside constant function detected.
Call trace: [Class.callback]. Aborting... J
EEP aborted

value: 10
```

The error message always shows the call trace unlike error message for constant variable violation because the offenders are member functions only in this case. This is transmissible, so if a constant function calls a non member function that changes the variable, it still generates runtime error. Even when the trace ends due to member functions invoking the offending non member functions, the partial trace should help with debugging.

3.3.2 The argnum & argnumvar Directives

These directives validate the number of arguments used to invoke the function. They take the count present in the declaration for reference. This feature goes against the grain of JavaScript but it is necessary for robustness.

The `argnum` directive considers the reference count to be exact, and generates runtime error when more or fewer arguments are used. The `argnumvar` directive considers the reference count as the minimum number that must be given but allows more arguments (the suffix `var` means variable). Both validations can be retained using the `pmc-argnum` flag.

```
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		value: 0,
		work__argnum: function(x, y){cout("working with",x,y)},
		workx__argnumvar: function(x, y){cout("working with",x,y,arguments[2])},
	}
})
let c = Class.New();
c.work(1,2);
c.workx(1,2,3)
try{c.work(33)}catch(e){cout(e)}
try{c.work(33, 100, 200)}catch(e){cout(e)}
try{c.workx(33)}catch(e){cout(e)}
```

```
working with 1 2
working with 1 2 3

JEEP: The function 'Class.work' was invoked with wrong argument count (declared: 2,
given: 1). Aborting...
JEEP aborted

JEEP: The function 'Class.work' was invoked with wrong argument count (declared: 2,
given: 3). Aborting...
JEEP aborted

JEEP: The function 'Class.workx' was invoked with wrong argument count (declared: 2,
given: 1). Aborting...
JEEP aborted
```

3.3.3 The argconst Directive

This directive renders the arguments of a function constant. Any attempt to modify them generates runtime error.

```
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		value: 0,
		work__argconst: function(x, obj){x = 0; obj.val = x},
	}
})
let c = Class.New();
try{c.work(33, {val: 100})}catch(e){cout(e)}
```

```
JEEP: The function 'Class.work' modified its arguments (arg.1) despite declaring them
constant. Aborting... JEEP aborted
```

The error simply gives the zero based index of the argument and not the name. This might be updated in the next version to issue names. Notice that only second argument is considered changed. This is because the first is a number, and as you know, they are passed by value unlike the second one which is passed by reference.

3.3.4 The argtype Directive

This directive enforces that the function is invoked only with arguments of the types as defined in the definition. Invoking with wrong types generates runtime error. Validating the argument count is a necessary step to validating the types, so this directive generates runtime error if the function is invoked with wrong number of arguments. It behaves like `argnum` rather than `argnumvar`. You can retain the validation in production mode with `pmc-argtype`.

This is another feature going against the grain of JavaScript but is necessary to for robustness. By enforcing types on arguments, it helps self documenting the code. At first this feature may seem too alien but a closer look at existing code written in plain JavaScript should prove otherwise. You will find code littered with `typeof` checks which is basically an imperfect and ad hoc mechanism to achieve what this feature delivers more comprehensively.

This is the most ambitious, complex and useful feature that also has a quirky syntax, so I will first show a minimal example.

```
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		value: 0,
		work__argtype: function(val__number){
			return function(val){
				cout("given value:",val)
			}
		}
	}
})
let c = Class.New();
c.work(10)
try{c.work({val: 100})}catch(e){cout(e)}
```

```
given value: 10

JEEP: The function 'Class.work' was invoked with wrong argument types
'val(number,*object)'. Aborting...
JEEP aborted
```

The argument type is indicated using the directive notation. In the error message the starred name indicates the wrong type that was used. It is reported this way because captures the intention while not being verbose. It also helps reporting all the errors in a single line concisely.

The inner function exists to circumvent a problem arising due to the nature of JavaScript. Unlike members with directives that can be processed by Jeep to separate the intended names to enable natural usage pattern, the arguments can't be put through that and are hard coded because there is no mechanism available in JavaScript that allows renaming arguments such that their names referenced in the code remains unaffected. This means, the argument inside the function will have the directive as part of its name. Therefore, the inner function exists to solve the problem. It is

returned because it is the actual function that will be executed. As I said, it is quirky. This is another way Jeep improvises to get things to work in a programmer friendly way.

The outer function must only return the inner function, that is there can't be any other code before or after the return statement. I must stress that the only thing inside the function is a single statement returning a function; you can't declare a function and return it in two statements. Not Anything else is considered syntax error.

The outer function must have at least one typed argument, otherwise it generates compile time error. This is enforced to keep the code meaningful: what is the purpose of a typed function if it tales only untyped arguments?

The inner function has to follow two rules. First, it must declare exact number of arguments as the outer function does. Second, the names of arguments must be exactly same as the undecorated names of the arguments used in the outer function. Breaking either rule generates compile time error.

```
try{DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		value: 0,
		work__argtype: function(val__number){
			return function(value){
				cout("given value:",val)
			}
		},
		work2__argtype: function(a,b){
			return function(a,b){}
		},
		work3__argtype: function(a__number){
		}
	}
})}catch(e){cout(e)}
```

```
JEEP: Compilation found 3 error(s) for class 'Class'.
1. The function 'work' with the 'argtype' directive has mismatched argument names.
2. The function 'work2' with the 'argtype' directive should have at least one typed
argument.
3. The function 'work3' with the 'argtype' directive doesn't have the correct function
definition.
JEEP aborted.
```

Jeep allows some flexibility by letting some arguments to be typeless. Such arguments can take any values as normal JavaScript code does, so these arguments are simply declared without any name decoration like normal JavaScript function arguments are.

```
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		value: 0,
		work__argtype: function(val__number, x){
```

```
                        return function(val, x){
                                cout("given value:",val, "x:",x)
                        }
                }
        }
})
let c = Class.New();
c.work(10,100);
c.work(10, "ten")
```

```
given value: 10 x: 100
given value: 10 x: ten
```

Jeep allows all the four JavaScript types, and also user defined types, which are essentially objects. They follow the same directive syntax.

- `v__number`: indicates the argument v is of `number` type
- `v__string`: indicates the argument v is of `string` type
- `v__array`: indicates the argument v is of `array` type
- `v__object`: indicates the argument v is of `object` type
- `v__myobj`: indicates the argument v is of a registered type `myobj`

The user defined type works with registered objects only because it is hard to represent created objects as types in the scheme that is followed. Note that the `EXTENDS` property uses an array, so it doesn't face the same problem.

With registered objects you can simply use the name. If the name have spaces or other such invalid characters from JavaScript syntax point of view, you can simply use the `Typedef` function to create a valid name. Further, the registered types are available globally to the application unlike the created ones making it easy to search for the types based on the name.

The user defined type can be registered after the definition of the function. It just has to be available at the time of execution of the function. This means, the argument can be of the same type as the class, if the class is registered.

Just so that you know, validating user defined type is done simply by calling the `InstanceOf` function with the argument; there is no other magic. Note that you need not do this test inside a typed function for an argument, unless you are doing in for an untyped argument, because the function is invoked only if the type test passes.

```
DemoEnv.RegisterClassDef("MyClass", {
        PUBLIC: {
                value: 0,
                work__argtype: function(val__number, rec__DataRecord, cl__MyClass){
                        return function(val, rec, cl){
                                cout("arg:",val,"rec.value:",rec.value,"c.value:",c.value)
                        }
                }
        }
})
```

```
DemoEnv.RegisterRecordDef("DataRecord", {value: 100})
let MyClass = JEEP.GetObjectDef("MyClass")
let c = MyClass.New();
let DR = JEEP.GetObjectDef("DataRecord")
c.work(10, DR.New(), c);
try{c.work(10, "ten", 10)}catch(e){cout(e)}
```

```
arg: 10 rec.value: 100 c.value: 0
JEEP: The function 'Class.work' was invoked with wrong argument types
'rec(DataRecord,*string) ,cl(MyClass,*number)'. Aborting...
JEEP aborted
```

Any type name used other than the four native ones is automatically considered user defined, and the associated object will be attempted to be retrieved and validated against. If such an object is not found it generates runtime error.

```
let Class = DemoEnv.CreateClassDef("MyClass", {
       PUBLIC: {
              value: 0,
              work__argtype: function(val__number, rec__FileRecord){
                     return function(val, rec){
                     }
              }
       }
})
let c = Class.New();
try{c.work(10, "ten")}catch(e){cout(e)}
```

```
JEEP: The argument type 'FileRecord' for the function 'MyClass.work' is unregistered.
Aborting...
JEEP aborted
```

Note that when a type is deemed unregistered, the validation process aborts immediately and the type mismatch error is not generated as validation doesn't reach that stage since it is pointless to continue to that stage.

3.3.5 The managed Directive

This directive renders the function valid to instantiate a managed class. This is related to constructor and destructor, and will be discussed in 3.6. This is not a robustness feature, but rather a core feature like virtual and abstract functions and is always enabled. This is transmissible.

3.4. Static Members

The static members are those that can be used without needing to instantiate the class. The consequence of this is that they are shared by code. Static members can be private or public only; there is no protected access privilege.. For simplicity, the access restrictions are applied via directives rather than due to declaration inside sub objects. By default static members are public. The access restrictions are controlled by the same `pmc-memaccs` flag.

The advantage of having non public static members is that instances can share them, especially variables, with no one other than other instances of the same type without having to resort to using global objects that are accessible to the entire application.

```
let Class = DemoEnv.CreateClassDef("Class", {
	CONSTRUCTOR: function(n){this.name = n},
	PUBLIC: {
		name: "",
		printPrivStat: function(){
			this.$static.printPrivStat(this.name)
		},
		changePrivStat: function(x){this.$static.privvalue = x},
	},
	STATIC: {
		pubvalue: 10,
		privvalue__private: 30,
		printPrivStat: function(name){
			cout(name+".privvalue:", this.privvalue)
		},
	}
})
let c1 = Class.New("first");
c1.printPrivStat();
c1.changePrivStat(1023);
c1.printPrivStat();
let c2 = Class.New("second")
c2.printPrivStat();
try{cout("Class.STATIC.pubvalue:",Class.STATIC.pubvalue)}catch(e){cout(e)}
try{cout("Class.STATIC.privvalue:",Class.STATIC.privvalue)}catch(e){cout(e)}
```

```
first.privvalue: 30
first.privvalue: 1023
second.privvalue: 1023
Class.STATIC.pubvalue: 10
JEEP: Attempt to read private static variable 'Class.privvalue' detected. Aborting...
JEEP aborted
```

Unlike instance members, static members are accessible in two forms intended for two locations. For non member functions, they are accessible via the `STATIC` property in the class definition. Within member functions, they are accessible via the `$static` property. It is advisable that you use this property and not `$def.STATIC` The reason will be apparent when static members is discussed in relation to hierarchy.

3.5. Constructor

A class constructor has more features than that of a structure discussed in 2.2, with everything mentioned there is applicable to class constructors.

3.5.1. Flexible Construction

A class is more flexible than a structure during instantiation. When instantiated with `New` and there is exactly one argument, class attempts `InitNew` automatically. If it is possible, then the instantiation behaves as if it was done with this function. If it isn't possible, class attempts copy construction. which is basically cloning a new instance using an existing one as its reference. If that is also not possible, the argument is simply forwarded to the constructor.

When copy construction happens the constructor is not invoked because the new instance simply takes everything from the reference instance that are already constructed, and there is no need for further construction. Any changes needed to member variables will be done by invoking member functions based on application logic.

```
let Class = DemoEnv.CreateClassDef("Class", {
	CONSTRUCTOR: function(x){
		if(this.$initnew)
			cout("InitNew mechanism invoked")
		else{
			cout("constructing...")
			this.value = x;
		}
	},
	PUBLIC: {
		value: 0,
		print: function(){cout("value:", this.value)}
	}
})
let c = Class.New({value: 100})
c.print();
c.value = 200;
Class.New(c).print();
Class.New(300).print();
```

```
InitNew mechanism invoked
value: 100
value: 200
constructing...
value: 300
```

3.5.2. Failed Construction

When a constructor throws an exception, the instantiation process is automatically aborted and an instance is not created at all. While this is the most obvious way to fail, a simpler way would be to directly return the reason of failure as a string from the constructor (non strings are converted to string with `toString`). This method must be used when the failure is a conscious one because it has better performance.. Construction failure has a lot more significance when classes have one or more bases and will be discussed along with the class hierarchy in a later chapter.

```
let Class = DemoEnv.CreateClassDef("Class", {
	CONSTRUCTOR: function(x){
		if(x < 0) return "negative value is invalid";
		this.value = x;
	},
	PUBLIC: {
		value: 0,
		print: function(){cout("value:", this.value)},
	}
})
try{Class.New(-10).print();}catch(e){cout(e)}
Class.New(10).print();
```

```
JEEP: The class 'Class' could not be instantiated. Reason: negative value is invalid.
Aborting...
JEEP aborted
value: 10
```

3.6. Managed Class and Destructor

Destructor is the counterpart of constructor. It is the last function called on an instance before the instance is destroyed. The actual destruction of the instance is left to the JavaScript engine, since it is a normal JavaScript object. But conceptually, an instance is destroyed as soon as the last destructor returns. I say last because classes with one or more bases an have more than one destructor, as you shall see when discussing class hierarchy.

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 within 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 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? How is the connection released? 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. It also allows us to be lazy, in that we can put the cleanup code inside 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 a mechanism that mimics all this.

The mechanism is imparted by the `managed` directive. Three links are needed to complete the mechanism ().

1. A class that needs this benefit uses this directive on the constructor. This is the only directive allowed for constructors. Such classes are called managed classes.
2. The managed classes defines a destructor.
3. The managed class is instantiated within a function that has this directive.

After this, irrespective of how the managed function exits, either due to return or throw, the destructors are invoked. Managed classes can be instantiated only within a managed function, or a chain started by it. Attempting it elsewhere generates runtime error.

Managed classes should define a destructor; not doing so is a compiler error. This is enforced to keep code meaningful: what is the purpose of all this setup if there is not destructor to be called at the end? Similarly, destructors can be defined only for managed classes. This is enforced in this direction as well to keep code meaningful: what is the purpose of a destructor if there is no mechanism to call it?

A managed class is instantiated just like a normal class. The significance of the managedness is in the destruction mechanism rather than the construction mechanism, other than the location of valid instantiation.

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

```
DemoEnv.RegisterClassDef("Logger", {
	CONSTRUCTOR__managed: function(){},
	DESTRUCTOR: function(){this.Commit()},
	PUBLIC: {
		Append: function(a){ this.logs.push(a)},
		Commit: function(){
			cout("Logging the following", this.logs.length,"entries")
			for(let k = 0; k<this.logs.length; k++)
				cout((k+1),this.logs[k])
		}
	},
	PRIVATE: {
		logs: []
	}
})
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		value: 0,
		work__managed: function(){
			let logger = JEEP.GetObjectDef("Logger").New()
			logger.Append("first")
			logger.Append("second")
			logger.Append("third")
			throw new Error("simulated exception")
		}
	}
})
try{Class.New().work()}catch(e){}
cout(“other code”)
```

```
Logging the following 3 entries
1 first
2 second
3 third
Error: simulated exception
other code
```

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.

A destructor cannot have any directives and return any value. Returning a value is not an error per se and they are simply ignored. However, a destructor is prohibited from throwing exceptions, and any attempt to do so generates runtime error. Jeep aborts because destructor are supposed to be invoked when scope ends, which is forced to happen when exceptions are thrown. Now, if the destructor itself throws exception, there is nothing else to catch it. This behavior is inherited from C++.

3.7. Strong and Weak Robustness

Strong robustness features are those that catch the evildoer in the act, stop the act and report it. In contrast, weak robustness features are those that are imparted after the fact by doing some investigation. They are still robust, but weak, and thus fragile in some scenarios. Weakness exists because Jeep is limited by what JavaScript offers as a language. Of course, everything could be made strong but that leads to either unnatural usage patterns or severely penalizing performance.

The robustness due to the directives `argnum`, `argnumvar` and `argtype` are strong. The access restriction is strong.

The constanstness related robustness features provided by `const` and `argconst` directives are weak in some scenarios. These are strong when dealing with native objects that are used and passed by value but weak when dealing with objects. The validation happens at the function exit. So, while the mutation is captured in the end, the mutated data can be transmitted around and there is no way to validate or stop this. So, an undesired effect could lead to wrong storing entry in the user database on the server, so beware!

Consider this example.

```
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC:{
		value: 0,
		info__const: {value: 0},
		printValue__const: function(){
			this.value = 100;
			cout("value:", this.value)
		},
		print: function(){
			this.info.value = 100;
			this.show();
```

```
            },
            show: function(){ cout("info.value:", this.info.value)}
        }
})
try{Class.New().printValue()}catch(e){cout(e)}
try{Class.New().print()}catch(e){cout(e)}
```

```
JEEP: Attempt to change the variable 'Class.value' inside constant function detected.
Call trace: [Class.printValue]. Aborting...
JEEP aborted
info.value: 100
JEEP: Attempt to change constant variables 'Class.info' detected. Call trace:
[Class.print]. Aborting...
JEEP aborted
```

Making all of them strong is a top priority issue for the next version and I am sure there is a solution to that if I think hard enough. After all, several things I had imagined to be of similar nature during the second iteration, which led to some pretty bad features and usage patterns, improved greatly in the third iteration, so I am hopeful.

4. Field and Group

Fields and groups are a bit different from records, structures and classes. They don't follow the object generation and instantiation way of working. Yet they have functions that register them, which adds to their quirkiness. They are sort of pre instantiated. They are meant to represent API which are a bch of functions.

4.1. Field

A field is an exact opposite of a record. While record are intended to be a collection of data only, this is intended to be a collection of functions only. A record must accommodate functions due to the nature of JavaScript which treats functions as data, but a field will not accommodate data at all. If you need them either capture them as a closure, or, better, use a group.

Field functions are all public by default; and you can't use access specifier like you do for structure functions. If you need such features either capture the funcgtions as a closure, or, better, use a group.

Field functions can have directives, but only those that are not meant for class instances. So the directives `const`, `virtual` and `abstract` are prohibited.

You cannot have a field without functions since it is infinitely absurd.

Attempting to do anything contrary to these will generates compile time error.

A field needs an object inside which to add functions. You may pass any object. Often, it is beneficial to use `window` so that you may use the functions directly in a script. In development mode, it is validated that the object doesn't already have any properties with the same name. If there are, a compiler error is generated. In production mode this validation is not done.

There are two field creation related functions.

```
CreateField: function(where, name, def){},
RegisterField: function(name, def){},
```

The `CreateField` function takes the object explicitly, whereas the `RegisterField` function uses an internal object. If `null` is passed for the object, the field is created inside the `window` object. A registered field is retrieved with the same `GetObjectDef` function as with other objects. Registering a field has the same benefit as registering other objects.

```
let obj = {};
DemoEnv.CreateField(obj, "Field", {
       square: function(x){return x*x},
       print: function(m,x){cout(m,x)}
})
let n = obj.square(10)
```

```
obj.print("square of 10:", n)
```

```
square of 10: 100
```

Though a field essentially adds functions to a given object, a field is superior to using function properties with plain JavaScript code because the field functions can have directives and provide you all the robustness features applicable

---

You can abuse a field to hack a structure. This shouldn't be surprising because a field is essentially a structure without data along with some class function directives. And unlike hacking a structure out of a record, this one will actually work because the functions are processed and directives applied, but it isn't advisable in order to keep things sane and sensible. Use a structure instead.

```
let obj = {value: 100};
DemoEnv.CreateField(obj, "Field", {
       print: function(){cout("value:", this.value)}
})
obj.print();
```

```
value: 100
```

4.2. Group

A group is similar to a field in most respects, and sort of a super set of it. There are two functions related to creating groups and they behave identically to the field counterparts. The only difference is what goes inside the declaration.

```
CreateGroup: function(where, name, def){},
RegisterGroup: function(name, def){},
```

You can think of a group as a stripped down class. This applies both to the declaration as well as behavior. The major difference it has with classes are

- groups cannot have static members
- groups cannot have protected members
- groups cannot be extended
- groups cannot have virtual and abstract directives
- groups cannot have managed constructor
- groups cannot have destructor

To redundantly mention the opposite of the bulleted list for sake of clarity, groups can have constructors, members with access restrictions and PMC flags.

A group can have data, and consequently it can define a constructor to do initialization. However, since it is pre instantiated in principle, the mechanism to invoke the constructor is via the `Init`

function that is added to the group object, rather the `New` or `InitNew`. In fact, these functions are unavailable to use for groups. Unlike structure and class where `New` cannot take arguments if a constructor is not defined, the Init itself is not available if the group has no constructor. It is designed this way because `New` is more fundamental to those objects while `Init` is only optional for a group, and incorporating the same logic was pointless.

```
DemoEnv.RegisterGroup("Maths", {
	CONSTRUCTOR: function(accuracy){
		cout("initializing settings...")
		this.accuracy = accuracy;
	},
	PUBLIC: {
		calculate: function(x){
			this.docalc(x);
		}
	},
	PRIVATE: {
		accuracy: 0.001,
		docalc: function(x){cout("calculating",x, "with accuracy",
							this.accuracy,"...")}
	}
})
let M = JEEP.GetObjectDef("Maths")
M.Init(0.01);
M.Init(0.003);
M.calculate("log(10)")
try{M.docalc("log(10)")}catch(e){cout(e)}
```

```
initializing settings...
calculating log(10) with accuracy 0.01 ...
JEEP: Attempt to invoke private function 'Maths.docalc' detected. Aborting...
JEEP aborted
```

The `Init` function is called only once no mater how many times it is invoked. I am not sure if multiple calls must be deemed an error; for now it isn't. What happens if you don't call this function is left to the group and Jeep doesn't try to impose anything by being too clever, not here at least.

5. Class Hierarchy

This chapter is about class hierarchies, and having read the Class chapter is the pre requirement.

Only classes can be part of a hierarchy, which is built as a consequence of inheritance. Inheritance is process through which a class acquires members from other classes in a disciplined way subject to validations. The acquiring class is called the derived class and the providing classes are called base classes. The classes that are distant bases are called ancestors. The class that has no bases but has derived classes is called root class. The class that has only bases classes and no derived classes are called leaf classes (no, a solitary classes without bases or derived classes is not its own base and derived class).

The essence of inheritance is two fold. Firstly, it's a clean way to reuse existing code. Inheritance is a system generated composition because internally that is what it amounts to. However, unlike user generated composition, system generated composition leads to natural usage patterns. The inherited members become the derived class' own members and so can be accessed naturally, as opposed to user generated composition where the inner object must be accessed first.

Secondly, classes inherit in the hope of retaining most of the inherited behavior and tweaking some other behaviors. This end result of such a tweaking is called 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++. a derived class providing its own implementation to bases class' virtual function is called overriding.

Inheritance is achieved by listing the bases in the `EXTENDS` property. This is similar to records but dialed up to a hundred. If the array has one base, it amounts to single inheritance and if there are more it amounts to multiple inheritance. 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 and then build on it to discuss multiple inheritance.

5.1. Single Inheritance

5.1.1. The Basics

A derived class gets everything from the base class though it can't access everything. The private members of the base is inaccessible and attempting to do so generates runtime error. Only public and private members are accessible by derived classes.

```
let Base = DemoEnv.CreateClassDef("Base", {
       CONSTRUCTOR: function(v){
              cout(this.$name,"base constructing...")
              this.value = v;
       },
       PUBLIC: {
```

```
                print: function(){
                        cout("base value:", this.value)
                }
        },
        PROTECTED:{
                value: 0,
        },
        PRIVATE: {
                privvar: 0
        }
})
let Derived = DemoEnv.CreateClassDef("Derived", {
        EXTENDS: [Base],
        CONSTRUCTOR: function(v){
                cout(this.$name,"derived constructing...")
        },
        PUBLIC: {
                show: function(){
                        cout("derived value:", this.value)
                        try{cout("value:", this.privvar)}catch(e){cout(e)}
                }
        }
})
let d = Derived.New(100);
d.print();
d.show();
```

```
Derived base constructing...
Derived derived constructing...
base value: 100
derived value: 100
JEEP: Attempt to read base class private variable 'Base.privvar' from derived class
function detected. Call trace: [Derived.show]. Aborting...
JEEP aborted
```

The constructors are called from root class to leaf, and whichever constructors are available are called along the way. All get the same arguments. Inheritance merges all base classes with the derived class. 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. You can see that the property `$name` resolves to the derived class' name both times. 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. Making it the C++ way is of course possible, but that leads to severe performance penalty, which is not judicious to bear for a feature of questionable utility.

Member names cannot repeat in the hierarchy. If they do, it generates compile time error. This is the basic restriction for inheritance and quite unlike C++. At one level, this is a consequence of the nature of JavaScript. At another level, this is my fixing something flawed in C++. Repeated member names makes no sense and causes a semantic nightmare. The solution in C++ is to use scope resolution, that is you explicitly mention the class name to indicate whose member you are trying to access. We could do something similar by moving clashing members to a sub object

inside the instance. and give it the base name, but it is not viable with Jeep because the classes are merged. Suppose F is the function repeating in both base B and derived D. Suppose we move it to the object B inside the instance d, making F accessible as `d.B.F()`. Now, if B had G that used F, and G is not repeated, then G exists in d and accessible as `d.G()`. In its implementation, G is going to use F as `this.F()`, which is non existent now. Well, depending on the classes, it might actually end up calling D's F instead of B's own and produce inadvertent and unintended polymorphic behavior, which basically breaks the code. So Jeep simply forbids repetition.

5.1.2. Failed Construction

In the series of constructors that are called as the part of instantiation of a derived class, even if one fails, instantiation is aborted immediately. If it is a managed class the destructors are called in reverse order, that is leaf to root.

If a base class is declared as managed, the derived class should also be declared managed. Not doing it generates compile time error. This basically falls through and enforces that all classes that have a managed class as base or ancestor declare their constructors as managed, and due to that, define a destructor.

```
let TopBase = DemoEnv.CreateClassDef("TopBase", {
	CONSTRUCTOR__managed: function(v){
		cout("topbase constructing...")
	},
	DESTRUCTOR: function(v){
		cout("topbase destructing...")
	},
	PUBLIC: {
		value: 0,
		print: function(){},
	},
})
let MidBase = DemoEnv.CreateClassDef("MidBase", {
	EXTENDS: [TopBase],
	CONSTRUCTOR__managed: function(v){
		cout("midbase constructing...")
	},
	DESTRUCTOR: function(v){
		cout("midbase destructing...")
	},
})
let LowBase = DemoEnv.CreateClassDef("LowBase", {
	EXTENDS: [MidBase],
	CONSTRUCTOR__managed: function(v){
		return  "invalid initial value given"
	},
	DESTRUCTOR: function(v){
		cout("lowbase destructing...")
	},
})
let Derived = DemoEnv.CreateClassDef("Derived", {
	EXTENDS: [LowBase],
	CONSTRUCTOR__managed: function(v){
		cout("derived constructing...")
	},
	DESTRUCTOR: function(v){
		cout("derived destructing...")
	},
```

```
})
try{Derived.New()}catch(e){cout(e)}
```

```
topbase constructing...
midbase constructing...
JEEP: The class 'Derived' could not be instantiated. Reason: invalid initial value
given. Aborting...
JEEP aborted midbase
destructing...
topbase destructing...
```

5.1.3. Polymorphism

The directive `virtual` renders a function polymorphic. A function once defined as virtual should always be defined as virtual throughout the hierarchy; not doing so generates compile time error. This is unlike C++ where the directive virtual an be omitted in derived class declarations. Jeep enforces this in order to draw attention to the polymorphic nature of the functions: what can be more important than a polymorphic function in a class hierarchy?

```
let Base = DemoEnv.CreateClassDef("Base", {
	PUBLIC: {
		value: 0,
		run: function(){
			this.print();
			this.work();
		}
	},
	PROTECTED:{
		print__virtual: function(){cout("base print")}
	},
	PRIVATE: {
		work__virtual: function(){cout("base work")}
	}
})
let Derived = DemoEnv.CreateClassDef("Derived", {
	EXTENDS: [Base],
	PROTECTED:{
		print__virtual: function(){
			cout("derived print")
			this.$base.Base.print();
		}
	},
	PRIVATE: {
		work__virtual: function(){cout("derived work")}
	}
})
let d = Derived.New();
d.run();
```

```
derived print
base print
derived work
```

Virtual functions are best kept protected or private because they are mostly going to be implementation details, but as with everything in software engineering, this is just a guideline rather than a rule.

The derived class can override virtual function irrespective of its access restriction, except that it can't invoke the overridden base function if it's a private one.

The overridden virtual functions are always available inside a sub object with the same name as the base class, which is inside the reflective property `$base`. As mentioned above, private virtual functions are not accessible, so they are not included in this object.

As mentioned, classes with unimplemented abstract functions cannot be instantiated. An abstract function cannot have definition, and must be empty. Even having a semicolon is considered non empty and compile time error is generated. Because abstract function are only specification and not implementation, these functions don't get added to the base object since they don't need to be, and frankly can't be, invoked.

```
let Base = DemoEnv.CreateClassDef("Base", {
	PUBLIC: {
		value: 0,
		run: function(){
			this.print();
			this.work();
		}
	},
	PROTECTED:{
		print__abstract: function(){}
	},
	PRIVATE: {
		work__abstract: function(){}
	}
})
let Derived = DemoEnv.CreateClassDef("Derived", {
	EXTENDS: [Base],
	PROTECTED:{
		print__virtual: function(){cout("derived print")}
	},
	PRIVATE: {
		work__virtual: function(){cout("derived work")}
	}
})
try{Base.New()}catch(e){cout(e)}
Derived.New().run();
```

```
JEEP: The class 'Base' cannot be instantiated due to presence of unimplemented abstract
functions 'Base.print,Base.work'.
Aborting... JEEP aborted

derived print
derived work
```

The `virtual` and `abstract` directives are not robustness related, but are core features like `managed`, so are always effective.

Polymorphism is disabled 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 (as hierarchy is written on paper) 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 in 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 the virtual function that it has implemented, 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 and not the overridden one. 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 TopBase = DemoEnv.CreateClassDef("TopBase", {
	CONSTRUCTOR__managed: function(v){
		cout("TopBase constructing...")
		this.tbprint();
	},
	DESTRUCTOR: function(v){
		cout("TopBase destructing...")
		this.tbprint();
	},
	PUBLIC: {
		value: 0,
		tbprint__virtual: function(){cout("TopBase printing...")},
	},
})
let MidBase = DemoEnv.CreateClassDef("MidBase", {
	EXTENDS: [TopBase],
	CONSTRUCTOR__managed: function(v){
		cout("MidBase constructing...")
		this.mbprint();
	},
	DESTRUCTOR: function(v){
		cout("MidBase destructing...")
		this.mbprint();
	},
	PUBLIC: {
		tbprint__virtual: function(){cout("MidBase printing...")},
		mbprint__virtual: function(){cout("MidBase printing...")},
	},
```

```
})
let Derived = DemoEnv.CreateClassDef("Derived", {
	EXTENDS: [MidBase],
	CONSTRUCTOR__managed: function(v){
		cout("Derived constructing...")
		this.print();
	},
	DESTRUCTOR: function(v){
		cout("Derived destructing...")
		this.print();
	},
	PUBLIC: {
		print__virtual: function(){cout("Derived printing...")},
		tbprint__virtual: function(){cout("Derived printing...")},
		mbprint__virtual: function(){cout("Derived printing...")},
	},
})
let Class = DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		v: 0,
		test__managed: function(){
			cout("<TopBase instantiated>")
			TopBase.New();
			cout("<MidBase instantiated>")
			MidBase.New();
			cout("<Derived instantiated>")
			Derived.New()
		}
	}
})
Class.New().test();
```

```
<TopBase instantiated>
TopBase constructing...
TopBase printing...
<MidBase instantiated>
TopBase constructing...
TopBase printing...
MidBase constructing...
MidBase printing...
<Derived instantiated>
TopBase constructing...
TopBase printing...
MidBase constructing...
MidBase printing...
Derived constructing...
Derived printing...
Derived destructing...
Derived printing...

other output edited for brevity
```

The bug that this kind of 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 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-disabled-virtual-call* flag in the environment. This aborts when such calls are detected.

When trapped from the constructor the instantiation is aborted. This is the only correct action that can be taken. If it is a managed class, then the usual destruction process ensues and behaves exactly a described earlier. The only change here is that exception is framework generated rather than application generated.

```
let Env = JEEP.CreateEnvironment({mode: "development-mode",
                      flags: "trap-disabled-virtual-call"})

let Base = Env.CreateClassDef("Base", {
       CONSTRUCTOR__managed: function(v){
              cout("base constructing...")
              this.print();
       },
       DESTRUCTOR: function(v){
              cout("base destructing...")
              this.print();
       },
       PUBLIC: {
              value: 0,
              print__virtual: function(){cout("base printing...")},
       },
})
let Derived = Env.CreateClassDef("Derived", {
       EXTENDS: [Base],
       CONSTRUCTOR__managed: function(v){
              cout("derived constructing...")
       },
       DESTRUCTOR: function(v){
              cout("derived destructing...")
       },
       PUBLIC: {
              print__virtual: function(){cout("derived printing...")},
       },
})
let Class = Env.CreateClassDef("Class", {
       PUBLIC: {
              v: 0,
              test__managed: function(){
                     Derived.New().print();
              }
       }
})
try{Class.New().test()}catch(e){cout(e)}
cout("other code")
```

```
base constructing...
JEEP: Invoking virtual function 'Base.print' in the constructor detected. Aborting...
derived destructing...
base destructing...
JEEP: Invoking virtual function 'Base.print' in the destructor detected. Aborting...
JEEP: The instance of class 'Derived' could not be destroyed since destruction was
aborted due to invalid virtual function call.
JEEP: DESTRUCTOR THROWING EXCEPTION IS A SERIOUS STRUCTURAL ERROR. Aborting...
JEEP aborted
```

```
other code
```

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 and 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.

```
let Class = DemoEnv.CreateClassDef("Class", {
	CONSTRUCTOR: function(callback){
		callback(this)
	},
	PUBLIC: {
		v: 0,
		print__virtual: function(){
			cout("printing...")
		}
	}
})
Class.New(function(inst){
	inst.print()
})
```

5.1.3. Static Members

Unlike instance members, static members are not merged, so different classes can use same names and yet be part of the hierarchy. Static members remain with the class and must accessed only via the appropriate objects.

The class' own static members are accessed with the `$static` property by instances as described in 3.4. If base classes have static members, they are accessible much like the overridden virtual functions are. They are also put inside `$base` object but this object is inside the `$static` object instead of the instance to keep things consistent. Similar to instance members, private base static members are inaccessible to derived classes, and any attempt to do so will generates runtime error.

```
let Class = DemoEnv.CreateClassDef("Class", {
	CONSTRUCTOR: function(n){this.name = n},
	PUBLIC: {
		name: "",
		printPrivStat: function(){
			cout(this.name+".privvalue:", this.$static.privvalue)
		},
		changePrivStat: function(x){this.$static.privvalue = x},
	},
	STATIC: {
		pubvalue: 10,
		privvalue__private: 30,
	}
})
let Derived = DemoEnv.CreateClassDef("Derived", {
	EXTENDS: [Class],
	PUBLIC: {
		work: function(){
```

```
                    cout(this.name+".privvalue:", this.$static.privvalue)
                    cout(this.name+".Class.pubvalue:",
                                        this.$static.$base.Class.pubvalue)
                    try{this.$static.$base.Class.privvalue = 0;}catch(e){cout(e)}
             }
      },
      STATIC: {
             privvalue__private: -1
      }
})
let c1 = Class.New("firstbase");
c1.changePrivStat(1023);
let d = Derived.New("derived");
d.printPrivStat();
d.work();
try{Derived.STATIC.privvalue = 0;}catch(e){cout(e)}
```

```
derived.privvalue: 1023
derived.privvalue: -1
derived.Class.pubvalue: 10
JEEP: Attempt to write private static variable 'Class.privvalue' detected. Aborting...
JEEP aborted
JEEP: Attempt to write private static variable 'Derived.privvalue' detected.
Aborting...
JEEP aborted
```

Even though the static members are not merged, the instances certainly are. So, member functions should not access static members via `this.$def.STATIC`, although it is available, to avoid bad results. If a class does this, and gets extended, then the attempt could end up either failing since the static member might not exist in the derived instance, or worse, it could end up using the derived instance's static member, thereby providing the absurd static polymorphism. Since the `$def` and the generated definition used by non members are actually the same object, the `STATIC` property is not removed. Instead, a more obvious `$static` property is provided which the programmer cannot ignore and go on to use stuff stuffed in a deeply nested object.

5.2. 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. It is somewhat like mixins but a lot more structured, disciplined and potent, and actually produces correct results.

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 this is the lack of understanding of 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.

5.2.1. 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, forming a rhombus. 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.CreateClassDef("TopBase", {
	CONSTRUCTOR__managed: function(){cout("TopBase constructing...")},
	DESTRUCTOR: function(){cout("TopBase destructing...")},
	PUBLIC: {v: 0, f: function(){}}
})
let MidBaseA = DemoEnv.CreateClassDef("MidBaseA", {
	EXTENDS: [TopBase],
	CONSTRUCTOR__managed: function(){cout("MidBaseA constructing...")},
	DESTRUCTOR: function(){cout("MidBaseA destructing...")},
})
let MidBaseB = DemoEnv.CreateClassDef("MidBaseB", {
	EXTENDS: [TopBase],
	CONSTRUCTOR__managed: function(){cout("MidBaseB constructing...")},
	DESTRUCTOR: function(){cout("MidBaseB destructing...")},
})
let BaseX = DemoEnv.CreateClassDef("BaseX", {
	CONSTRUCTOR__managed: function(){cout("BaseX constructing...")},
	DESTRUCTOR: function(){cout("BaseX destructing...")},
	PUBLIC: {vv: 0, ff: function(){}}
})
let Derived = DemoEnv.CreateClassDef("Derived", {
	EXTENDS: [MidBaseA, MidBaseB, BaseX],
	CONSTRUCTOR__managed: function(){cout("Derived constructing...")},
	DESTRUCTOR: function(){cout("Derived destructing...")},
})
DemoEnv.CreateClassDef("Class", {
	PUBLIC: {
		v: 0,
		test__managed: function(){
			Derived.New();
		}
	}
}).New().test();
```

```
TopBase constructing...
MidBaseA constructing...
MidBaseB constructing...
BaseX constructing...
Derived constructing...
Derived destructing...
BaseX destructing...
MidBaseB destructing...
MidBaseA destructing...
TopBase destructing...
```

In everyday English, you can think of the bases being different lines of inheritance. The construction starts at the top of each line, starting from the line on left and moving rightwards. The destruction process follows exactly the opposite order in both dimensions. The order of listing the bases has consequence only to construction and destruction and nothing else. Of

course, the same order is applicable during failed construction but I won't show another long and bring example. It simply partially destructs the lines in the destruction order.

5.2.2. Reinforcing Abstraction

This is mostly meaningful in hierarchy that have diamonds. It could be a conscious design decision or just a consequence of using multiple bases. When one of the base classes in the hierarchy declares an inherited virtual function as abstract, it is considered reinforcement of abstraction. When done consciously, it can 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 how reinforcement has no effect. So Jeep simply establishes a rule – reinforcement stays even if one base does it.

```
let TopBase = DemoEnv.CreateClassDef("TopBase", {
	PUBLIC: {
		v: 0,
		work__virtual: function(){}
	}
})
let MidBaseA = DemoEnv.CreateClassDef("MidBaseA", {
	EXTENDS: [TopBase],
	PUBLIC: {
		work__virtual: function(){}
	}
})
let MidBaseB = DemoEnv.CreateClassDef("MidBaseB", {
	EXTENDS: [TopBase],
	PUBLIC: {
		work__abstract: function(){}
	}
})
let Derived = DemoEnv.CreateClassDef("Derived", {
	EXTENDS: [MidBaseA, MidBaseB],
})
try{Derived.New()}catch(e){cout(e)}
```

```
JEEP: The class 'Derived' cannot be instantiated due to presence of unimplemented
abstract functions 'MidBaseB.work'. Aborting...
JEEP aborted
```

5.2.3. Problems and Solutions

As with any topic, people who dislike multiple inheritance put a lot of subjectivity into it which could be a combination of poor understanding of the topic and poor orientation to the topic. That said 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? A base itself might have multiple bases and might 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).

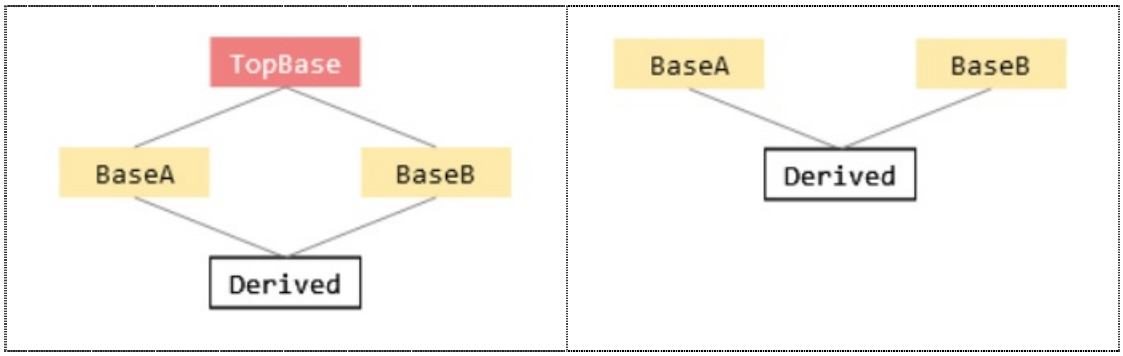
The simplest solution is to ban duplicate names like Jeep does. While this is beneficial for single inheritance, it is crucial for multiple inheritance. This rule solve most of the problem single-handedly. The other solution is to use wrapper classes, explained in 5.3.5.

5.2.4. Valid and Invalid Hierarchies

Consider these hierarchies. They are screen shots from a simple hierarchy visualizer tool that is demonstrated in Example chapter. 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 virtual, gray indicates it must be implemented, blue indicates plain member, and white indicates absence of the function.

H1

H2



H3

H4

The hierarchy H1 is invalid due to the ground rule, and H2 is invalid for the same reason but also 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 H3 and H4 respectively.

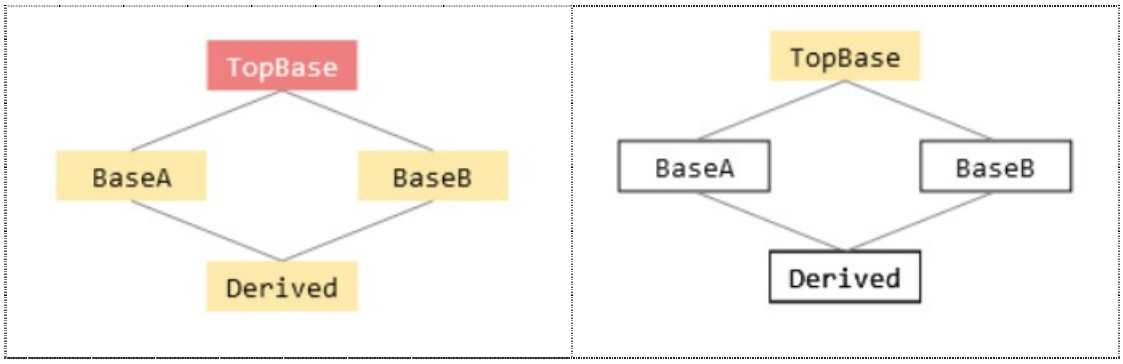
As with duality, it is an error for the same function to be virtual in multiple bases. While both are virtual, unlike duality, there is a still a semantic error. Just because `BaseA` and `BaseB` both have a virtual function with same name, it doesn't mean the function has the same meaning or is used 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). 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 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. With this, 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 in the above "solutions", but it is a solution chosen by the programmer based on analysis, hopefully, rather than an auto generated solution with a `one-size-fits-all` approach. This is what Jeep enforces, the programmer intervention that is, not the one size approach.

The following scenarios are valid.

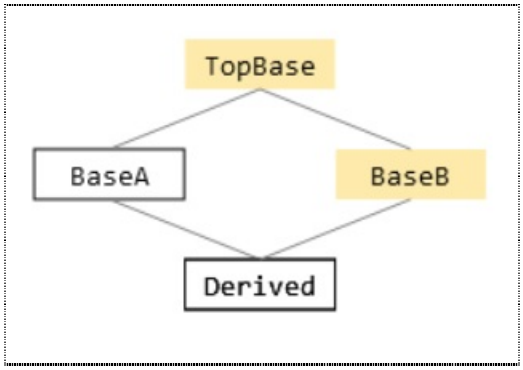


H5 H6

The hierarchy H5 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.

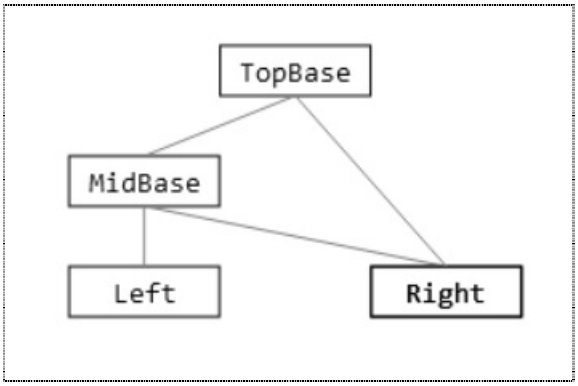
In H6, both `BaseA` and `BaseB` inherit the same function which is inherited by the `Derived`. There is no real clash because the derived class receives the same function from multiple sources.

However, the following hierarchy, which is a variation H6, 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.



H7

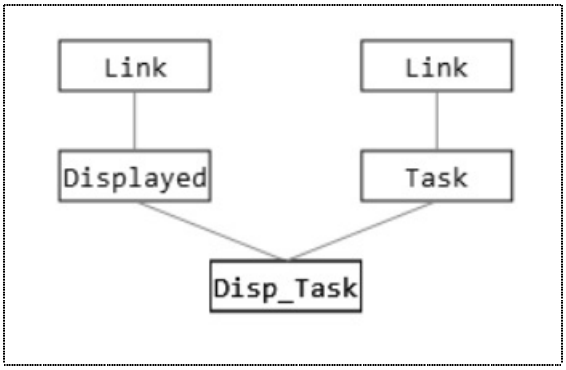
The following structure is also invalid in Jeep due to the derived class `Right`, though it is valid in C++, and probably other languages. Note that all boxes are white, meaning this structure is invalid even if none of the classes have `SomeFunction`.



H8

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.

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.



H9

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. The 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. I don't see any other valid use cases for this shape Further, this is one scenario where composition is natural, preferable and actually the 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.

5.2.5. Wrapper Class

Wrapper classes exist to solve the name clash problem in multiple inheritance. They can be used in single inheritance also, but, as you shall see, they are relevant only in multiple inheritance.

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 robbing the chance to use this wonderful technique. So Jeep offers wrapper classes as the solution.

The obvious solution to the problem is to change the offending names. This is acceptable only if the classes are not going to be used anywhere other than the current hierarchy because name clash is really hierarchy dependent. You would have to modify the source code multiple times to include the class in multiple hierarchies, which is a bad solution beyond comprehension. Further, considering that you might have not control of sources of all class that you want to use but can't, changing the sources quickly becomes infeasible even if your manager thinks its an apt solution.

To reflect a bit on this problem, it is actually a ramification of the naming rule established by Jeep., but it is a problem worthy of having and solving in order to build semantically correct and robust code. The best part the rule is that it allows understanding the hierarchy, its behavior and even its inner working by simply inspecting a 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.

The solution to the problem is still renaming and nothing else, but it must be done keeping in mind a lot of things. It is one of those things that looks simple to solve but has immense amount of intricacies to consider. Wrapper classes solve the problem effectively by tacking all the intricacies. They are created with this function.

```
CreateClassWrapper: function(className, desc)
```

The argument `className` is the name of the registered class to be wrapped. Wrapping works only with registered classes because such classes can be expected to be third party or unfeasible to be modified, unlike the local ones created which can be readily changed if name clash happens and don't need wrapping. The argument `desc` is an object that simply provides a map of old name to new name for both variables and functions. Note that wrapping works only on instance members and not static members because static members are not merged and hence don't cause name clash.

The function returns a class definition where renaming is done correctly. This definition is supposed to be used in place of the wrapped class. Wrapper classes cannot be instantiated, and generate runtime error if attempted to, because their instances serve no purpose. Further, wrapping a wrapper is not possible because the wrapped class must be registered but the generated wrapper class is always craated.

Of all objects, wrappers by far have the simplest descriptor object but are the most complex in terms of behavior and implementation. The behavior is more about not doing the wrong thing, so the examples show how wrappers behave transparently no matter what kind of classes are wrapped.

Wrapping works irrespective of access restrictions, and it maintains the restrictions.

```
let BaseX = DemoEnv.CreateClassDef("BaseX", {
	PUBLIC: {
		callPrint: function(){this.print()},
	},
	PROTECTED: {
		print: function(){cout("BaseX.value:", this.value)}
	},
	PRIVATE: {
		value: 1,
	},
})
DemoEnv.RegisterClassDef("BaseY", {
	PUBLIC: {
		value: 2,
		print: function(){cout("BaseY.value:", this.value)}
	}
})
let WrBaseY = DemoEnv.CreateClassWrapper("BaseY", {
	Functions: {
		"print": "show"
	},
	Variables: {
		"value": "number"
	}
})
let Derived = DemoEnv.CreateClassDef("Derived", {
	EXTENDS: [BaseX, WrBaseY],
	PUBLIC: {
		test: function(){
			this.print();
			this.callPrint();
			this.show();
			cout("derived number", this.number);
			try{this.value}catch(e){cout(e)}
		}
	}
})
let d = Derived.New();
d.test();
d.callPrint();
try{d.value}catch(e){cout(e)}
```

```
BaseX.value: 1
BaseX.value: 1
BaseY.value: 2
derived number 2
JEEP: Attempt to read base class private variable 'BaseX.value' from derived class
function detected. Call trace: [Derived.test]. Aborting...
JEEP aborted
BaseX.value: 1
JEEP: Attempt to read private variable 'BaseX.value' detected. Aborting...
JEEP aborted
```

The biggest triumph of wrapper class is that for the class deriving from the wrapper class and every code downstream, only the new names are available, yet, the original names are accessible to all classes above it. This is achieved without changing the source code, which would be impossible with simple renaming.

I call this situation the FGK problem: if a function (or a variable) F is renamed to G and there exists a non renamed function K that uses F, the renaming must not break the code. As you can see, the need to rename and the FGK problem are in direct opposition to each other and both must be solved. Wrapper classes solve this tricky problem quite effectively.

Wrapping also preserves polymorphism.

```
let BaseX = DemoEnv.CreateClassDef("BaseX", {
	PUBLIC: {
		callPrint: function(){this.print()},
	},
	PROTECTED: {
		print__virtual: function(){cout("BaseX.value:", this.value)},
		value: 1,
	},
})
DemoEnv.RegisterClassDef("BaseXX", {
	EXTENDS: [BaseX],
	PUBLIC: {
		print__virtual: function(){cout("BaseXX.value:", this.value)}
	}
})
let WrBaseXX = DemoEnv.CreateClassWrapper("BaseXX", {
	Functions: {
		"print": "show"
	},
})
let BaseY = DemoEnv.CreateClassDef("BaseY", {
	PUBLIC: {
		count: 2,
		print: function(){cout("BaseY.value:", this.count)}
	}
})
let Derived = DemoEnv.CreateClassDef("Derived", {
	EXTENDS: [BaseY, WrBaseXX],
	PUBLIC: {
		test: function(){
			this.print();
```

```
                    this.callPrint();
                    this.show();
                }
        }
})
Derived.New().test();
```

```
BaseY.value: 2
BaseXX.value: 1
BaseXX.value: 1
```

While wrappers look elegant, and behave flawlessly, there is an enormous performance penalty that has to paid in return. More about this can be read in the implementation chapter, but this part must be kept in mind. The more wrapper bases you have in a hierarchy, the worse the performance gets, so use them sparingly as a solution of last resort. If the offending classes are your own, just change them if it is feasible. Using wrappers for this would be highly inefficient, something like looping an array and adding it to an empty array instead of calling `Array.from`.

Two simple tips can prevent unnecessary performance overhead by avoiding wrappers.

1. If you think you need a wrapper class, don't just assume but verify. Try inheriting and see if there are name clashes.
2. If you do use wrappers, and also do tweaking and refactoring of the base classes, try to use the bases directly to check if names clash. You might have forgotten what names you renamed, and if those still exist after refactoring etc.

You have to keep in mind that wrappers are a workaround, and like any workaround they are less than ideal but quite necessary.

5.3. A Critique of Multiple Inheritance

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 this being equated to an evil thing 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 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 already done it a few posts below.

Their main argument is a slippery slope and highly 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 inheritance mechanism.

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.

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 twisting base ends of a wire to light a bulb reasoning that the switch anyway does the same thing in principle under the hood.

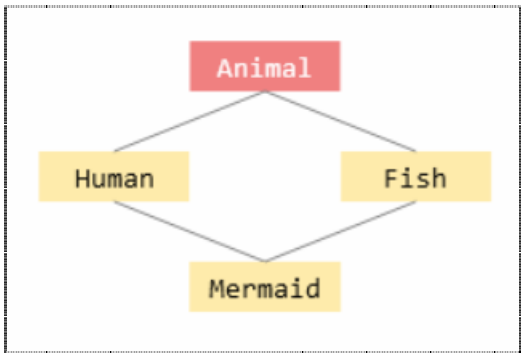
As for their other argument, I guess calculus must be reconsidered as a compulsory topic taught in engineering since most students just don't get it.

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 multiple inheritance 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 in understanding multiple inheritance.

A class can be seen as composed of attributes and action, represented by variables and functions respectively. When mixing is restricted to attributes, 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, automation means having variables for each of such attributes with `get` and `set` directives, and using `InitNew` to instantiate 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. The related classes are shown in 1.3.



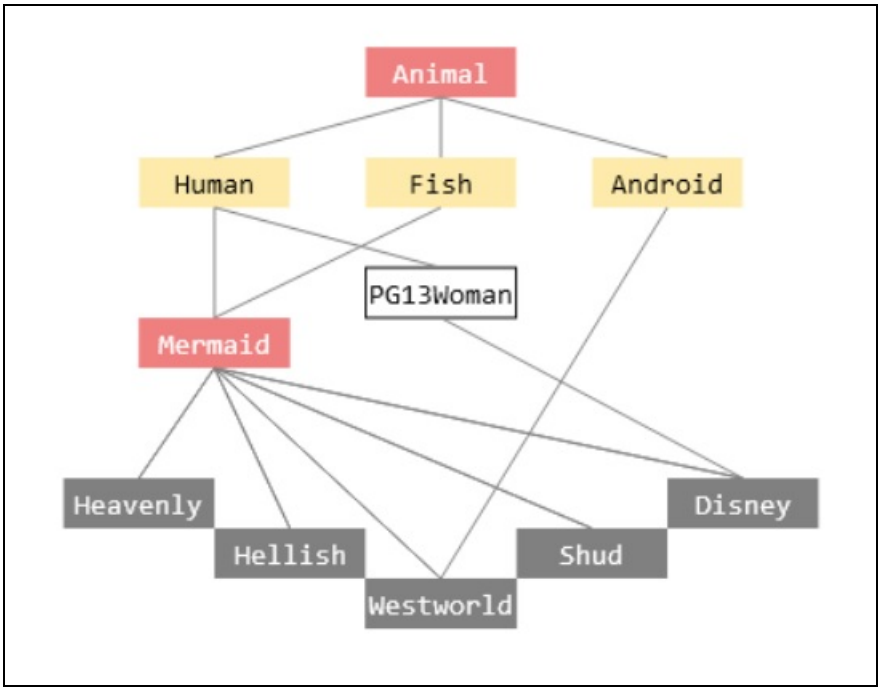
As one might expect, the implementation of `Mermaid` would look like this.

```
DemoEnv.RegisterClassDef("Mermaid", {
	EXTENDS: ["Fish", Human],
	PROTECTED: {
		breathe__virtual: function(){this.$base.Human.breathe()},
		move__virtual: function(){this.$base.Fish.move()},
	}
})
```

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 such as *before, after* etc is not going to be useful in any scenario other than the one concocted in the demonstration code shown for such a system.

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

For the proposed game, 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 a class hierarchy 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 so in this hierarchy.

You might then have some code like this to let the fun begin.

```
let mmarr = [
    JEEP.GetObjectDef("MMHeavenly").New(),
    JEEP.GetObjectDef("MMHellish").New(),
    JEEP.GetObjectDef("MMWestWorld").New(),
];
for(let k = 0; k<mmarr.length; k++)
    mmarr[k].liveOneMoment();
```

6. Namespace and Library

These two are organizational objects rather than functional ones with which you structure your code and objects for better presentation and usage. These two are elated to each other, in that a library internally uses a namespace. So these are explained together.

6.1. Namespace

Something like namespace is necessary because Jeep allows registering objects. Since registered objects are stored in a common database, there is bound to be name clashes. As you might expect, Jeep disallows registering more than one object with the same name. Namespaces provide a solution to this in a simple and elegant way. The name is taken directly from C++ where it solves a somewhat similar problem.

A namespace is created using `CreateNamespace` function present in the environment object. Contrary to what you might have expected, the function takes no arguments. You don't have to supply a name to help with the disambiguation because there is no guarantee that that name will be unique. As they say, it would be turtles all the way down. So, the necessary disambiguation will be based on internally generated information.

A namespace being a name disambiguation mechanism contains all the API present in the environment object that deal with names. It also contains the object access and query functions of the main object since they deal with names as well. The object looks like this.

```
Namespace = {
	GetEnvironment: function(){},
	RegisterRecordDef: function(name, def){},
	RegisterStructDef: function(name, def){},
	RegisterClassDef: function(name, def){},
	RegisterField: function(name, def){},
	RegisterGroup: function(name, def){},
	GetObjectDef: function(name){},
	ObjectDefExists: function(name){},
	Partition: function(names){},
	Flatten: function(){},
}
```

The idea is to create a namespace and create objects using it rather than directly with the main object. This way, if everyone creates a namespace to create their objects, it is guaranteed that none of the names clash, exception of course the clash within the namespace itself. If this is not due to programmer laziness but a genuine sticky situation like the original one, then there is a solution for that as well that will be explained a bit later. Since a namespace is created with an environment object, the properties of environment apply automatically.

You use these functions on a namespace object as you do with the environment and the main object, that is you use the names directly without having to care about the disambiguation since that is taken care of internally. The `GetEnvironment` function exists to help know who created the namespace, and also allow creation of local objects.

6.1.1. The Basics

```
DemoEnv.RegisterStructDef("MyStruct", {
       value: 0,
       print: function(){cout(this.$name,this.value)}
})
let NS = DemoEnv.CreateNamespace();
NS.RegisterStructDef("MyStruct", {
       value: 1,
       print: function(){cout(this.$name,this.value)}
})
JEEP.GetObjectDef("MyStruct").New().print();
NS.GetObjectDef("MyStruct").New().print();
```

```
MyStruct 0
MyStruct 1
```

As you can see, the usage is so identical that you can literally switch the environment with a namespace if you suddenly realized that you need to avoid name clashes. Notice that the names of the instances are unaffected.

The only thing with namespace is that the user of the registered objects must have access to the namespace, which means, the registered objects are available to only those who have access to the namespace where they were registered. This is not bad at all, but actually beneficial because it essentially creates a private data store for objects that can be shared with only a select few. This gives you the benefit of both creating and registering objects.

The other seemingly bad part about namespace is that you cannot know if you are using an instance from the main store or the private store by just looking at the name. But as with many superficial issues, this one quickly vaporizes once you realize that this is actually good because that is what transparency means: why would you care where the instance came from as long as it behaves as advertised? Further, identical names mean nothing more than identical name, so such non issues should not stop you from using namespaces.

6.1.2. Typedef

The `Typedef` function really shines when used in conjunction with namespaces. In fact, doing this is the only way you can access the objects defined in a namespace without having to refer to the namespace. This might seem countering what namespace exist for, but it has a purpose. Consider this code to see what this means.

```
DemoEnv.RegisterStructDef("NotMyStruct", {
       value: 0,
       print: function(){cout(this.$name,this.value)}
})
let NS = DemoEnv.CreateNamespace();
NS.RegisterStructDef("MyStruct", {
       value: 1,
       print: function(){cout(this.$name,this.value)}
})
JEEP.GetObjectDef("NotMyStruct").New().print();
let NSStruct = NS.GetObjectDef("MyStruct");
NSStruct.New().print();
```

```
JEEP.Typedef("StructT", NSStruct.$name)
let ST = JEEP.GetObjectDef("StructT");
let st = ST.New();
st.print();
cout("Implementation name of",st.$name,":",ST.$name)
```

```
NotMyStruct 0
MyStruct 1
MyStruct 1
Implementation name of MyStruct : .ns1.MyStruct
```

The shown code is pointless, its for demonstration only. The internal name is shown for demonstration purposes only, so don't rely on this implementation to be retained.

The real power of this mechanism is that it allows doing argument type validation when the types are created in namespaces and libraries. This is the only reason to use this function, and the only reason it exists.

```
let MyClass = DemoEnv.CreateClassDef("MyClass", {
	PUBLIC: {
		value: 0,
		work__argtype: function(val__number, rec__MyDataRecord){
			return function(val, rec){
				cout("given value:",val, "rec.value:",rec.value)
			}
		}
	}
})

let NS = DemoEnv.CreateNamespace();
NS.RegisterStructDef("DataRecord", {
	value: 1,
	print: function(){cout(this.$name,this.value)}
})

JEEP.Typedef("MyDataRecord", NS.GetObjectDef("DataRecord").$name)

let c = MyClass.New();
let DR = NS.GetObjectDef("DataRecord")
c.work(10, DR.New());
```

```
given value: 10 rec.value: 1
```

Notice that the argument type uses the name crated using `Typedef` while the instance given as argument is of the original definition retrieved from the namespace.

6.1.3. Partitioning and Flattening

The 'native' functions of a namespace are partitioning and flattening. These are orthogonal operations, and are of great use in structuring your objects and accessing them respectively.

As the name indicates, `CreatePartition` partitions the namespace. The argument it takes is a comma separated names list which is assigned to each partition. There would as many partitions

as names. Duplicate names will generate runtime error in development mode only. There is no way to retain this validation in production mode as it seemed a bit too much.

A partitioned namespace will have a sub object created named '$', and all the partitions will be accessible on it as normal object properties. The partitions themselves are namespaces, so you can create a nice tree structure with repeated partitioning.

```
let ns = DemoEnv.CreateNamespace();
ns.Partition("first, second, third");
cout("Main namespace parts:", Object.keys(ns.$).join(','))

let second = ns.$.second;
second.Partition("top, bottom")
second.$.top.Partition("x,y,z")
cout("Sub namespace parts:", Object.keys(ns.$.second.$).join(','))
```

```
Main namespace parts: first,second,third
Sub namespace parts: top,bottom
```

Although a tree is a natural organizational structure, it can get pretty laborious to access the parts, since every level needs to go through a dollar sub object. The solution is to flatten the namespace. The previous example can be extended as shown.

```
let ns = DemoEnv.CreateNamespace();
ns.Partition("first, second, third");
cout("Main namespace parts:", Object.keys(ns.$).join(','))

let second = ns.$.second;
second.Partition("top, bottom")
second.$.top.Partition("x,y,z")
cout("Sub namespace parts:", Object.keys(ns.$.second.$).join(','))

let flat = ns.Flatten();
cout("Flattened namespace parts:",Object.keys(flat).join(','))

cout("Main namespace parts:", Object.keys(ns.$).join(','))
```

```
Main namespace parts: first,second,third
Sub namespace parts: top,bottom
Flattened namespace parts: first,x,y,z,top,bottom,second,third
Main namespace parts: first,second,third
```

The function takes no arguments and returns an object containing key-value pairs, where keys are the names and values are the sub namespaces. It doesn't alter the internal layout of the namespace itself. The flattening is done in depth first post order mechanism where all the descendents appear before the parents.

Since flattening doesn't alter the internal structure of namespaces, a nested namespace could either be accessed directly from the object returned by the function, or via a sub namespace using

the dollar objects. In the above example, the namespace `x` could be accessed in two ways: `flat.x` and `flat.second.$.x`.

In development mode, flattening throws exception if duplicate names exist. This is quite possible since different sub namespaces can have sub namespace with same name. There is no way to retain this validation in production mode as it seemed a bit too much.

6.2. Library

A library is a higher level organizational structure that builds on namespaces. Actually, there is no library object at all; what exists is just a namespace. However, the namespace offers enough abstraction and functionalities to be considered a separate object it its own right. In this day and age when identity is a touchy issue, who are we to judge if a library is really a namespace?

As with other objects, a library is usually hacked using functions and closures in plain JavaScript code. Jeep makes it structured and refined by providing a neat interface. The idea is that the provider registers a library and the consumer retrieves it and uses it. Jeep provides a very convenient mechanism to achieve all this.

6.2.1. Registering and Retrieval

The `RegisterLibrary` function present in the main object does the needful. The function takes two arguments. The first is the name. The second is the constructor function. Unlike other objects, a library takes a function directly; there is no other way to do what libraries intend to do in JavaScript other than separating the code indo a function, unless one concocts and elaborate scheme just to make it seem rich and complex.

Registering libraries has the same benefits as registering other objects have, because unlike the plain JavaScript code, this function is not a globally available function that can be used by any code, but rather to only those who go looking for it specifically, and cannot be used accidentally.

A registered library is retrieved using the `GetLibrary` function in the environment object. It invokes the constructor, but only once when the library is retrieved for the first time. Subsequent usage of this function simply returns the initialized library. The arguments of the function are not processed but forwarded to the constructor.

The function calls the constructor with a namespace object added to the `this` object. A library is supposed to use this namespace as its private store. Note that it can still the main object, but the objects pertaining to the library are better created with the namespace provided to make the code meaningful.

```
JEEP.RegisterLibrary("MyLibrary", function(x){
       cout("initializing", this.$name, "with", x)
       this.namespace.RegisterRecordDef("Rec", {value: x*100})
});
let Lib = DemoEnv.GetLibrary("MyLibrary", 3);
Lib = DemoEnv.GetLibrary("MyLibrary", 12);
let r = Lib.GetObjectDef("Rec").New();
cout("record.value", r.value);
```

```
initializing MyLibrary with 3
```

```
record.value 300
```

Note that the `RegisterLibrary` function is in the main object but the `GetLibrary` function is in the environment object. It is due to the automatic availability of the namespace inside the constructor that the retrieval function is in the environment object. The environment creates and attaches the namespace to the `this` object. The registration is just about mapping a library constructor to a name, but the actual construction happens during retrieval. The constructor doesn't have to return anything, and the return value is ignored.

6.2.2. Building

Most libraries are going to have low two digit number of objects, and some of them would need to capture some closures. all this can be certainly done inside a single constructor but for better management, you would split the activity into several helper functions. However, these helpers must then become global, and this takes us back to square one, the square which Jeep wants to skip. So it offers a mechanism that avoids that square. The mechanism is called building the library. The this object of the constructor has two functions: `Build` and `BuildPrivate` for the purpose. The latter is a special case of the former, and I will explain the former first.

The helper functions are called builders, and they must be first registered for a library using the `RegisterLibraryBuilder` function in the main object. This function takes three arguments. the first is the name of the builder, the second is the name of the library and the third is the constructor for the builder. The library constructor can then automate calls to these functions via the `Build` function.

```
JEEP.RegisterLibrary("MySmallLib", function(x){
       cout("initializing", this.$name, "with", x)
       this.Build({
              "": {builder: "sbbldr", args: x+3},
              "sublib": {builder: "sbbldr", args: x+5},
       })
})

JEEP.RegisterLibraryBuilder("sbbldr", "MySmallLib", function(x){
       cout("initializing", this.$name, "with", x)
       this.namespace.RegisterRecordDef("Rec", {value: x*100})
})

let Lib = DemoEnv.GetLibrary("MySmallLib", 3);
let r = Lib.GetObjectDef("Rec").New();
cout("record.value", r.value);
r = Lib.$.sublib.GetObjectDef("Rec").New();
cout("record.value", r.value);
```

```
initializing MySmallLib with 3
initializing MySmallLib/sbbldr with 6
initializing MySmallLib/sbbldr with 8
record.value 600
record.value 800
```

The `Build` function is quite a versatile function. It takes an object whose keys are the names of the sub namespaces to be generated by partitioning. The partitioned namespace would be setup as

the namespace inside the `this` of the builder named, which will be invoked with the arguments mentioned. The arguments is optional, of course. Empty key indicates that the namespace itself must be used instead of the sub namespace.

The name of the builder is setup in a simple path notation where the root has the library's name. In real application code, the builders would be in separate files.

As you can see, the building mechanism is superior to what you can do with plain JavaScript. It is safe because it doesn't expose the builders as global functions, It is flexible because it allows creating a structure that best suits the logical grouping of objects for the library. It is productive because you can automate the process which can be controlled trivially.

The power and flexibility of the `Build` function is further generalized by the `BuildPrivate` function. The "private" in the name suggests that you can cause the building to happen on a locally created namespace inside the function, which it returns as result, instead of the one that constitutes the library. This allows libraries to have implementation details, where they can pick and choose which components they must expose by using the returned namespace as the reference data store. Except for this part, the function is exactly same as the `Build` function.

This, rather large, example tries to show something real that can be done with building. It shows the client code first, then the result and after that the library implementation.

```
let Lib = DemoEnv.GetLibrary("Lib", {gr3d: true});
Lib = Lib.Flatten();
let socket = Lib.network.GetObjectDef("Socket");
let s = socket.New();
cout("socket created for: ",s.protocol);

let CF = Lib.graphics.$.gr3d.GetObjectDef("CoolFeatures")
let pt3d = Lib.gr3d.GetObjectDef("Point");
let pt = pt3d.InitNew({x: 10, y: 10});
try{CF.DoCF(pt)}catch(e){cout(e)}
try{CF.DoAnotherCF(pt)}catch(e){cout(e)}
```

```
initializing Lib ...
socket created for: tcp/ip
DOING VEDNOR GRAPHICS FEATURE at 10 10 0
CoolFeatures.DoAnotherCF not available.
```

```
JEEP.RegisterLibrary("Lib", function(options){
	cout("initializing library...")
	this.Build({
		"graphics": {builder: "grBuilder", args: options},
		"network": {builder: "nwBuilder"}
	})
})
JEEP.RegisterLibraryBuilder("nwBuilder", "Lib", function(){
	this.namespace.RegisterRecordDef("Socket", {protocol: "tcp/ip"})
})
JEEP.RegisterLibraryBuilder("grBuilder", "Lib", function (options){
	let builderInfo = {};
```

```
	if(options.gr2d)
		builderInfo["gr2d"] = {builder: "gr2dBuilder"}
	if(options.gr3d)
		builderInfo["gr3d"] = {builder: "gr3dBuilder", args: options}
	this.Build(builderInfo);
})
JEEP.RegisterLibraryBuilder("gr2dBuilder", "Lib", function (options){
	this.namespace.RegisterRecordDef("Point", {x: 0, y: 0})
})
JEEP.RegisterLibraryBuilder("gr3dBuilder", "Lib", function (options){
	let ns = this.BuildPrivate({
		"native": {builder: "native3dLib"},
		"vendor": {builder: "vendor3dLib"}
	});
	this.namespace.RegisterRecordDef("Point", {x: 0, y: 0, z: 0})
	this.namespace.RegisterField("CoolFeatures", {
		DoCF: function(point){
			execFeature("DoCF", point)
		},
		DoAnotherCF: function(point){
			execFeature("DoAnotherCF", point)
		},
	})
	function getFeature(lib, field, func){
		if(!ns.$[lib].ObjectDefExists(field))
			return null;
		let fieldObj = ns.$[lib].GetObjectDef(field)
		return fieldObj[func];
	}
	function execFeature(feature, args){
		let func = getFeature("native", "CoolFeatures", feature)
		if(!func)
			func = getFeature("vendor", "CoolFeatures", feature)
		if(!func)
			throw "CoolFeatures."+feature+" not available."
		return func.apply(null, [args]);
	}
})
JEEP.RegisterLibraryBuilder("native3dLib", "Lib", function(){
	this.namespace.RegisterRecordDef("Point", {x: 0, y: 0, z: 0})
})
JEEP.RegisterLibraryBuilder("vendor3dLib", "Lib", function (){
	this.namespace.RegisterRecordDef("Point", {x: 0, y: 0, z: 0})
	this.namespace.RegisterField("CoolFeatures", {
		DoCF: function(point){
			cout("DOING VEDNOR GRAPHICS FEATURE at", point.x,point.y,point.z)
		},
	})
})
```

7. Utilities

The `Utils` object inside the `JEEP` object contains some functions and structures that an application could use. These are used in Jeep's own implementation, so they are tested in real world application. They are also formally tested along with Jeep, so the applications can use these with good amount of trust.

The object looks like this.

```
Utils = {
	CopyProps: function(src, dest, propNames){},
	SplitTrim: function(t, c){},
	MakeFlags: function(id, fnames, env){},
	RecursiveFlag: struct{},
	ObjectIterator: struct{},
	FlagProcessor: struct{},
	MessageFormatter: struct{}
}
```

7.1. CopyProps

This function copies properties from one object to another. It takes three arguments – source object, destination object and list of property names. The third is optional. When not given, all source properties are copied.

```
let dest = {};
JEEP.Utils.CopyProps({a: 1, b: 2, c: 3}, dest)
let keys = Object.keys(dest)
for(let k = 0; k<keys.length; k++)
	cout(keys[k], dest[keys[k]])
```

```
a 1
b 2
c 3
```

```
let dest = {};
JEEP.Utils.CopyProps({a: 1, b: 2, c: 3}, dest, ["a","b"])
let keys = Object.keys(dest)
for(let k = 0; k<keys.length; k++)
	cout(keys[k], dest[keys[k]])
```

```
a 1
b 2
```

During copying, it checks if the properties had been defined, in which case it defines it in the destination with the same property instead of simply copying to preserve the definition.

```
let dest = {};
```

```
let src = {a: 1, b: 2}
Object.defineProperty(src, "a", {
	get: function(){return -1}
})
JEEP.Utils.CopyProps(src, dest);
let keys = Object.keys(dest)
for(let k = 0; k<keys.length; k++)
	cout(keys[k], dest[keys[k]])
```

```
a -1
b 2
```

Despite this little bit of intelligence, this is a simple function and doesn't validate if destination already has the properties, nor does it see if the source has it for that matter when the third argument is given. It's the programmers responsibility to use this correctly.

7.2. SplitTrim

This function takes a text and the delimiting character as its arguments and returns an array of words that have been tokenized and trimmed of leading and trailing white spaces. This is a simple function and takes only one delimiter.

```
let text = " first, second,  third  ";
let st = JEEP.Utils.SplitTrim(text, ',')
// cout with concat to avoid spurious spaces
cout("input: \"" + text + "\"")
cout("output: \"" + st.join() + "\"");
```

```
input: " first, second,  third  "
output: "first,second,third"
```

7.3. MakeFlags

This function is quite useful when you have more than a few numerical flags to create which are orable. Unlike in C++, where flags are typically created using the #define preprocessor which results in a simple textual substitution, in JavaScript we have to create an object and assign values to its properties like this.

```
let Flags = {
	first: 1,
	second: 2,
	third: 4
}
```

This can get pretty tedious and error prone if it goes past 256. This function is intended to solve the problems by automating the flag creation process

The function takes three arguments. The first is the name to identify the flags as a whole. This is only useful when errors message is issued. The second argument is a string containing comma

separated names for the flags. The third is an environment object, and is optional. Errors are issued only when this is given and it is for development mode.

The third argument is needed for logistic reasons. All compile time validations in Jeep are always done in development mode and skipped in production mode, but to do so there is the environment object inside which the functions are called which has the mode. However, `Utils` exist separately, and it would be structurally unsound to add them inside environment. Hence this "dependency injection" is necessary.

The function internally uses `SplitTrim` to tokenize the names. It then creates an object, which it returns, and creates properties within it, each bearing the tokenized name in the given order. The values are automatically assigned powers of two starting from one.

```
let flags = JEEP.Utils.MakeFlags("test", "first, second, third")
let keys = Object.keys(flags)
for(let k = 0; k<keys.length; k++)
	cout(keys[k], flags[keys[k]])
```

```
first 1
second 2
third 4
```

Jeep issues a warning when 32 or more flags are attempted to be created. This is because the properties are integers and 2^{32} might cause overflow. A typical modern machine can be expected to have at least 32 bit word size, and since the first flag starts with one, 31 is the minimum number of safe flags that can be generated on modern machines of unknown word size.

```
let env = JEEP.CreateEnvironment({mode: "development-mode"});
let fnames = "";
for(let k = 0; k<33; k++)
	fnames += "f"+k+","
JEEP.Utils.MakeFlags("Test", fnames, env);
```

```
JEEP WARNING: Generating 32 or more flags (Test) might cause overflow on your machine.
```

Jeep aborts when flags are attempted to be created with repeated names.

```
let env = JEEP.CreateEnvironment({mode: "development-mode"});
let fnames = "";
try{JEEP.Utils.MakeFlags("Test", "f,g,f,g,f", env)}catch(e){cout(e)}
```

```
JEEP: The flag 'Test' has repeated flag names 'f,g'. Aborting...
JEEP aborted.
```

Jeep issues warning for overflow and doesn't abort since it is too cumbersome to check for 64 bit machines via environment or extra argument, and too draconian and shortsighted to abort. The programmer is expected to look for this warning during development.

7.4. RecursiveFlag

This is a small structure that is intended to be used to mark and unmark flags within reentrant functions. This is used extensively within the framework to do runtime validations. Read the **Internals** chapter for more details.

The structure looks like this. It also has a constructor but that is not part of the interface so it won't be discussed here.

```
RecursiveFlag = {
	add: function(){},
	isSet: function(){},
	remove: function(){},
}
```

The usage is pretty simple. You call `add` to add the flag, call `set` to check if the flag has been added, and call `remove` to remove the flag. All these are invoked on an instance. As you can see, none of the functions take any arguments. Only the set function returns a value, a Boolean to indicate the presence of the flag, and the other functions return nothing.

```
let rf = JEEP.Utils.RecursiveFlag.New();
cout("flag is", rf.isSet()?"set":"not set")
rf.add();
cout("flag is", rf.isSet()?"set":"not set")
rf.remove();
cout("flag is", rf.isSet()?"set":"not set")
for(let k = 0; k<10; k++)
	rf.remove();
rf.add();
cout("flag is", rf.isSet()?"set":"not set")
```

```
flag is not set
flag is set
flag is not set
flag is set
```

Every `add` should have a corresponding `remove` in order to balance the setting. However, you can remove more number of times than you add, and after that a single `add` will set the flag again. The removing operation is made idempotent after being unset as it is useful to lazy programmers.

The flag here is not same as the flags you create with `MakeFlags`. Unlike those flags which are numerical properties within an object assigned values in powers of two, this is a conceptual flag. You must have one instance per flag you want. For instance, Jeep uses one to validate constantness and one to validate access permission.

Internally, the flag is just a counter that is set to zero in the constructor. It is then incremented in `add` and decremented in `remove`. The flag is considered set if the counter is positive.

7.5. ObjectIterator

This is a structure intended to iterate over the properties of an object. The usual way of doing it via `Object.keys` and then looping the returned array becomes cumbersome when multiple objects are to be iterated in sequence or in nested fashion due to the explosion of the arrays and loop counters. The repeated array notation to access the keys and property values also gets tiring. This structure is intended to ease these things by providing a clean, higher level interface.

Caveat: Since this is a higher level structure, this will always be slower than the usual method because, unlike the usual method that works with simple, native data types, this uses a lot of function calls. However, this is noticeable only when iterating an object with thousands of properties inside a performance critical loop. In most other situations, the difference should be unnoticeable in the broader scheme of things.

Due to the caveat, Jeep uses this only during compilation, and only on objects that are expected to be small. At runtime, iteration is always in the usual method so as to get maximum performance.

The structure looks like this.

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

The general usage pattern is as follows:

1. instantiate the structure with the object whose properties you need to access,
2. call `GetNext` inside a while loop conditional
3. call `GetCurrPair` inside the while loop body
4. use the `key` and `value` which are properties of the returned pair

```
let iter = JEEP.Utils.ObjectIterator.New(JEEP.Utils.MakeFlags("","a,b,c,d,e"))
cout("total flags:", iter.Total())
while(iter.GetNext()){
	let pair = iter.GetCurrPair();
	cout(pair.key,pair.value)
}
```

```
total flags: 5
a 1
b 2
c 4
d 8
e 16
```

The function `Reset` exists to allow reusing the iterator instance. Internally, the constructor simply calls this function. The functions `GetCurrKey` and `GetCurrValue` exist in order to allow direct, one off access to the respective properties without having to create a pair object first.

Calling `GetNext` is not necessary if you need to access the first property only since `Reset` will always retrieve it if one exists.

This structure doesn't throw any exceptions; it chooses to ignore invalid inputs. The following example illustrates the behavior with different kinds of inputs and usage pattern.

```
// empty object
let iter = JEEP.Utils.ObjectIterator.New({})
cout("total properties in empty object:", iter.Total())

// direct access after reset
iter.Reset({a: 0, b: 1})
cout("direct access",iter.GetCurrKey(), iter.GetCurrValue())

// undefined object
iter.Reset();
cout("total properties in undefined:", iter.Total())
cout("key:", iter.GetCurrKey()||"undefined", "value:",iter.GetCurrValue()||"undefined")
while(iter.GetNext()){
	let pair = iter.GetCurrPair();
	cout("pair",pair.key,pair.value)
}

// non object
iter.Reset(100);
cout("total properties in integer:", iter.Total())
cout("key:", iter.GetCurrKey()||"undefined", "value:",iter.GetCurrValue()||"undefined")
while(iter.GetNext()){
	let pair = iter.GetCurrPair();
	cout("pair",pair.key,pair.value)
}
```

```
total properties in empty object: 0
direct access a 0
total properties in undefined: 0
key: undefined value: undefined
total properties in integer: 0
key: undefined value: undefined
```

7.6. FlagProcessor

This is a small structure intended to help with processing flags given as comma separated string and converting them to corresponding numerical values after doing all the necessary validations.

Note that the flags used here are not same as the ones created with `MakeFlags`. While that function converts a list of names into flags with values in powers of two, this function takes a list of flag names and returns a number or'd with the flag values which ought to be in powers of two. The two functions are orthogonal.

The structure looks like this.

```
FlagProcessor = {
	CONSTRUCTOR: function(flagMap){},
	Process: function(info){},
}
```

The constructor argument is a map of flag name to flag value. This is used as the reference for the processing that will be done. The values must be orable since that is what `Process` returns.
The `Process` function takes an object with these properties as its input argument.

```
info = {
	flags: “”,
	singleOnly: Boolean,
	markError: Boolean
}
```

The `flag` is the string that must be processed. The `singleOnly` flag indicates whether the input can have more than one flag. The `markError` flag indicates if the invalid flags must be marked and returned.

The function returns an object with two properties: `flags` and `errors`. The `flags` property is a number indicating all the flags that were present in the input. It is generated by or'ing the corresponding flag values. The `errors` property is created only if the input had invalid flags and the `markError` flag was set. If the `singleOnly` flag is set and the input contains more than one valid flag, the function returns `null`.

```
let fp = JEEP.Utils.FlagProcessor.New({
	first: 1,
	second: 2,
	third: 4
})
let res = fp.Process({flags: "first, second"})
cout("flag value:",res.flags)

res = fp.Process({flags: "first, second, fifth, sixth", markError: true})
cout("invalid flags:",res.errors.join(','))

res = fp.Process({flags: "first, second", singleOnly: true})
cout("result object:",res==null?"null":"not null")

res = fp.Process({flags: "fifth, sixth", markError: true, singleOnly: true})
cout("invalid flags:",res.errors.join(','))

res = fp.Process({flags: "first,second,fifth,sixth", markError:true, singleOnly:true})
cout("result object:",res==null?"null":"not null")
```

```
flag value: 3
invalid flags: fifth,sixth
result object: null
invalid flags: fifth,sixth
result object: null
```

The function internally uses `SplitTrim`.

When both `markError` and `singleOnly` are set and there are more than one valid input flags, irrespective of how many invalid flags exist, the function always returns `null`.

For the constructor argument, you can use flags created with `MakeFlags` instead of hard coding the numbers to avoid all the problems that the function exists to solve. Jeep does so in its implementation.

7.7. MessageFormatter

This is a small but extremely useful and powerful utility which does some nifty string processing. This is intended to be used to generate messages using templates. The underlying principle is similar to dynamically generating html pages with server side scripting.

The structure looks like this.

```
MessageFormatter = {
       CONSTRUCTOR: function(map){},
       Get: function(id, tags){},
}
```

The constructor argument is a map of the message id to the message template text. The templates will have place holders which will be filled during the generation stage. The `Get` function does the message generation. The argument `id` is supposed to be one of the ids present in the input map. The argument `tags` is an object whose keys are supposed to be the placeholders in the template and the values are supposed to be the data to be used to generate the message. The function returns the generated message. If the id was not found, the function returns `null`.

```
let MF = JEEP.Utils.MessageFormatter.New({
   "greetings": "Hello. My name is $last-name$, $first-name$ $last-name$.",
    "dollar": "$ means dollar",
    "-test": "Testing $dollar$ -count-.",
});

let m = MF.Get("test", {
    count: "1,2,3",
})
cout(m);

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

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

```
Testing $dollar$ 1,2,3.
```

```
Hello. My name is Povlatsky, Olya Povlatsky.
$ means dollar
```

By default, placeholders are determined using the dollar character. Any sequence of characters within two dollar characters is considered to be a placeholder. This can be overridden by prefixing the id with a symbol, which replaces the dollar character to mark the placeholder, but only for that message. Doing this is useful when you need to have dollar braced string to not act as place holder but also need a placeholder. The prefix is processed away, so its not part of the id given to `Get`. Unbalanced dollars are printed as is. A placeholder can appear any number of times and all will be replaced with the same value. If the `tags` don't contain the intended placeholder values, the message is generated with `undefined` as its value.

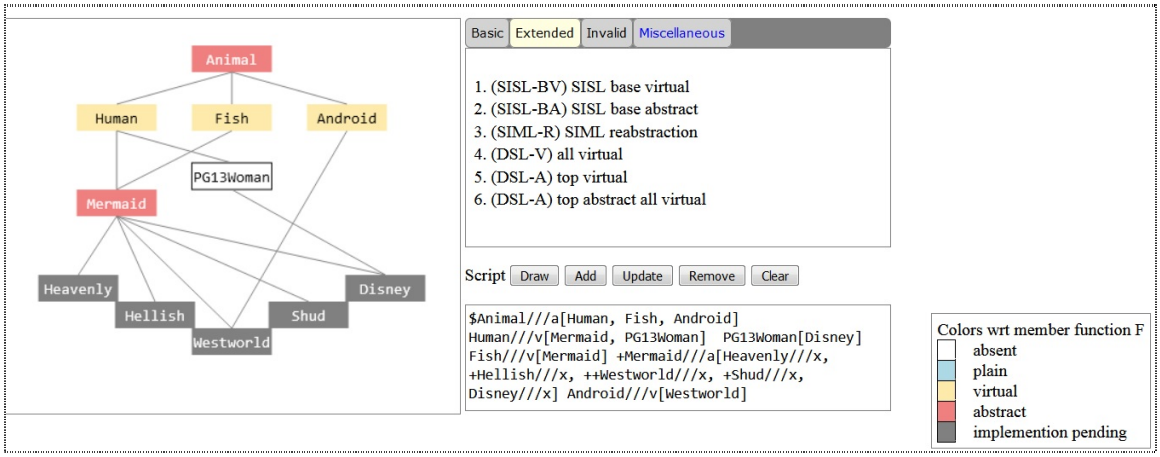
8. Example

This chapter exists because 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 sample. The distinguishing feature of a sample, as I describe it, is that it has a general quality of being useful but doesn't consider all scenarios or does all validations. It will attract at least one person who would want to improve it, and take it to the next level.

8.1. The Application

The application is a small hierarchy visualization tool. It is built using the Tiny library, which is part of the demonstration code. The tool is also part of the demonstration code. I created this tool in order to help me visualize various class hierarchies and reason about their correctness. The hierarchies discussed in earlier chapters were generated with this tool. I wanted something scriptable and automated, and didn't want to spend hours on MS Paint only to get substandard result at the end, so I create this. Another reason, and probably the primary reason, I created this was for engineering gratification – there is something profoundly gratifying about building self referencing things; here this tool is built using Jeep only in order to help improve Jeep. I call it "soft bootstrapping". Yet another reason for creating this was not to miss the chance to create a beautiful GUI application. I have always been a GUI guy and building GUI applications is something for which I can always make time and space.

8.1.1. The User Interface



As you can see, there are five components – diagram, tabs, script box, buttons and legend.

The legend is a static component generated by one of the classes. The diagram is a canvas. The tab is a set of html DOM elements managed by a class. The script is a simple text area.

Each tab contains a list of pre generated shapes except the last one which contains shapes generated from user scripts. Clicking on an item in each tab renders the shape on the canvas and also displays the script that generated it. Yellow background color indicates that the tab is active. Blue text marks the tab in which the currently displayed shape script exists. Within the tab, the

item representing the script will also be shown in blue. The buttons do whatever the names suggest. Draw only draws, Add adds to user scripts and also draws, Update updates the user script and also draws. Ctrl+Tab key combination is equivalent to clicking the Draw button.

8.1.2. The Script

The script is very simple. It is designed to be a regular language so that it can be easily processed with a regex; I didn't want to waste time doing text processing for a sample application.

The general pattern is `parent[child1, child2, ...],` where each entry is a name printed in the box. The names must not contain spaces.

The name itself follows this pattern `<ypos><root><dup>name<args>` and everything in angle brackets are options.

The `ypos` is one or more plus signs, intended to move a box downwards if the space cramps. Every plus moves it a constant step. I didn't want to do intricate positioning intelligence for a sample, so left it to the user to do the necessary positioning.

The `root` is a single dollar character that indicates that the box will be a root box. There can be multiple roots.

The `dup` is a single dot that allows multiple boxes with the same name. Boxes and names have one to one correspondence. A name occurring multiple times refer to the same box, so this was needed for flexibility.

The `args` are forward slash separated extra information called arguments. The first argument is always the background color, and the second is always the text color. The colors are RGB values given as 6 hexadecimal digits. The third is a single character, and can be one of the following and sets the box with the associated property.

- a abstract
- v virtual
- x to be implemented
- m plain member

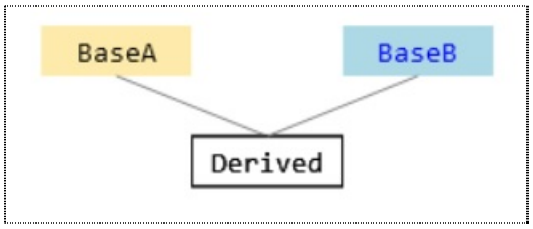
If a valid property is set, the application will override the colors set by other arguments. The arguments can be empty, but the order has to be maintained. So if you want to set only properties, you must have three slashes with the property appearing at the end.

Since a repeating name always refers to the first box that the name created, colors and properties set later will override the ones set earlier.

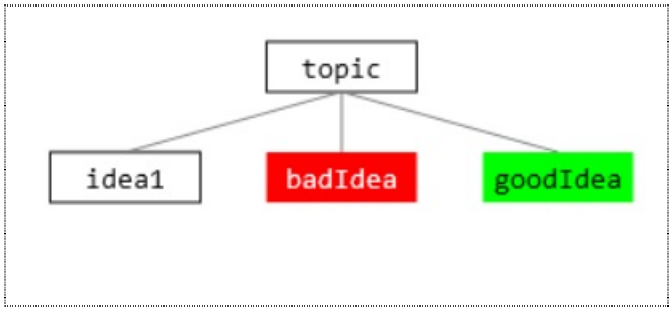
Note that a name which is not a root and doesn't have a parent will not appear in the diagram, though it will be internally created.

Three examples are shown below. You may play around with the script but remember that this is a sample only and has limited set of validation and features. For one, circularity is not tested. I leave it to the browser to complain of too much recursion.

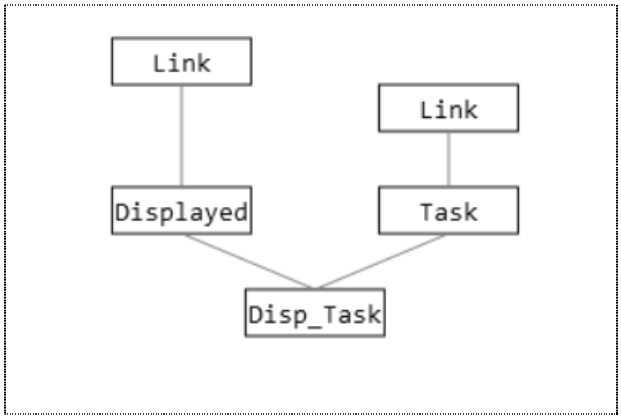
```
$BaseA///v[Derived] $BaseB///m[Derived]
```




```
$topic [idea1, badIdea/ff0000/ffffff, goodIdea/00ff00]
```




```
$Link[Displayed] +$.Link[Task] Displayed[Disp_Task] Task[Disp_Task]
```



8.2. The Code

The diagram canvas is managed by `TinyCanvas` class but the actual painting is done using `TinyPainter` class that uses the canvas class. The tabs is implemented by `TinyTab` class. The script box is a simple text area with no management necessary. The script processing is handled by the `TinyHierTree` class that creates the hierarchy that is shown on the canvas. This class also produces the legend. Each of these classes is in a separate library. A script in the application html page brings them all together.

The essence of Jeep is enabling writing intuitive, robust and well organized code. The actual implementation detail is less important because even ordinary JavaScript code composed of scattered set of functions can incorporate the same details and produce applications, as they have

been doing all this while. Therefore, only class and library outline will be shown and not implementation details.

8.2.1. TinyCanvas

This class either creates a new canvas or manages one given to it when instantiated. This class draws lines, boxes and texts, It saves and restores graphics properties during painting.

```
JEEP.RegisterLibrary("TinyCanvas", function(){
	this.namespace.RegisterStructDef("Point", {
		x: 0,
		y: 0,
		GetDistanceFrom: function(other){}
	})

	this.namespace.RegisterRecordDef("TextProperty", {
		font: "14pt Verdana",
		color: "black",
		align: "center",
	})

	this.namespace.RegisterClassDef("CanvasXY", {
		CONSTRUCTOR: function(canvas){},
		PUBLIC: {
			canvasElement__get: null,
			Clear: function(){},
			DrawLine: function(start, end, color){},
			DrawBox: function(topleft, rightbottom, color){},
			DrawText: function(at, text, prop){},
		},
		PRIVATE: {
			saveProps: function(){},
			restoreProps: function(){}
		}
	})
})
```

8.2.2. TinyPainter

This class internally uses a canvas. It simply instantiates one with the argument given to the constructor, without caring if its creating a new one or managing an existing one; it just needs a canvas to paint on.

This class paints boxes, lines and text and invokes the corresponding functions of the canvas to get the job done. However, this class provides a different interface for these operations. Unlike the canvas class where all properties are given as separate arguments, this class takes records as arguments that contain all the shape settings. This is much simpler and more high level.

Since painter is the one that will be really used to render shapes, this interface is much more useful and intuitive. For instance, rotating a line just needs changing a couple of members.

```
JEEP.RegisterLibrary("TinyPainter", function(){
	let Item = this.env.CreateRecordDef("Item", {
		x: 0,
		y: 0,
```

```
	})
	this.namespace.RegisterRecordDef("Line", {
		xa: 0,
		ya: 0,
		xb: 0,
		yb: 0,
		color: ""
	})
	this.namespace.RegisterRecordDef("Rectangle", {
		EXTENDS: [Item],
		width: 0,
		height: 0,
		fillColor: "",
		lineColor: "",
	})
	this.namespace.RegisterRecordDef("Text", {
		EXTENDS: [Item],
		content: "",
		color: "",
		font: "",
	})

	let TCLib = this.env.GetLibrary("TinyCanvas")
	let Canvas = TCLib.GetObjectDef("CanvasXY");

	this.namespace.RegisterClassDef("Painter", {
		CONSTRUCTOR: function(canvas){
			this.canvas = Canvas.New(canvas);
		},
		PUBLIC: {
			GetCanvasElement:function(){},
			Reset: function(){},
			AddLine: function(line){},
			AddRectangle: function(rect){},
			AddText: function(text){}
		},
		PRIVATE: {
			canvas: null,
		}
	})
})
```

8.2.3. TinyTab

This class basically shows and hides DOM elements given as tabs. It has a protected virtual function `UpdateTab` which is invoked via the public `UpdateAllTabs` function. The virtual function is protected to avoid accidental misuse. What this function does and when the update is called is application dependent. In this application, it is used to indicate active script tab with blue text by a derived class.

```
JEEP.RegisterLibrary("TinyTab", function(){
	this.namespace.RegisterRecordDef("Panel", {
		title: "",
		domElement: null,
	})

	JEEP.Typedef("TabPanel", this.namespace.GetObjectDef("Panel").$name)
```

```
	this.namespace.RegisterClassDef("Tab", {
		CONSTRUCTOR: function(){},
		PUBLIC: {
			AddPanel__argtype: function(panel__TabPanel){},
			ActivatePanel: function(pos){},
			SetSize: function(width, height){},
			UpdateAllTabs: function(){},
		},
		PROTECTED: {
			UpdateTab__virtual: function(tab){},
		},
		PRIVATE: {
			domElement__get: null,
			createTab: function(title){},
			// more
		},
	})
})
```

8.2.4. TinyHierTree

This class takes a script as input and produces the diagram if the script is valid. It uses `TuinyPainter` to paint. There is a virtual function called `GetNodeColors` that is invoked If there are arguments other than colors, since they are intended to be application specific. The function takes one argument, which is a slash separated argument set, which is everything other than the first two arguments which are hard coded in meaning. In this application, as mentioned earlier, these arguments are single letters, and they color the boxes as the legend indicates.

```
JEEP.RegisterLibrary("TinyHierTree", function(){
	let PainterLib = this.env.GetLibrary("TinyPainter")
	let Line = PainterLib.GetObjectDef("Line")
	let Rectangle = PainterLib.GetObjectDef("Rectangle")
	let Text = PainterLib.GetObjectDef("Text")

	let flags = JEEP.Utils.MakeFlags("", "NONE,MEMBER,VIRTUAL,ABSTRACT,IMPLEMENT")
	flags.NONE = 0;

	let Node = this.env.CreateRecordDef("Node", {
		name: "",
		children: [],
		// more
	})

	let LevInfo = this.env.CreateRecordDef("LevT", {nodes: [], y: -1})

	this.namespace.RegisterRecordDef("NodeColors", {
		backgroundColor: "white",
		textColor: "black",
	})

	this.namespace.RegisterClassDef("Tree", {
		CONSTRUCTOR: function(bw, script){},
		PUBLIC: {
			Paint: function(painter){ },
			Reset: function(script, painter){},
		},
```

```
		PROTECTED: {
			GetNodeColors__virtual: function(args){}.
		},
		PRIVATE: {
			createNode: function(build, name){},
			processName: function(n){},
			// more
		}
	})
})
```

8.2.5. Application Code

The application code is a script inside the html page. The code should be self explanatory if you have read about the components of Tiny. The only significant things are two: the `HierTab` class which extends the `TinyTab`, and the `AppHierTree` class which extends the `TinyHierTree` class.

The `TinyTab` class is a general purpose one that deals with tab management. The application has other things to do, such as render the clicked script, indicate active script etc. which are handled by the derived class. The class does some of these things in the overridden `UpdateTab` function. The management of user scripts – adding, updating and removing - is also done by this class and provides interface for it which are called by the button `onclick` handlers (not shown).

The `TinyHierTree` class is a general purpose one that processes scripts and produces the diagrams. This application has some specific requirements, so the class is extended by the `AppHierTree` class. It implements the `GetNodeColors` virtual function and returns the appropriate colors based on the arguments it gets. Other than this, the class provides a static function to generate the legend. The function is made static because the color information is fixed and not instance related.

```
JEEP.InitFramework();
Env = JEEP.CreateEnvironment({client: "jeep-aware", mode: "development-mode"})
//Env = JEEP.CreateEnvironment({client: "jeep-aware", mode: "production-mode"})

PainterLib = Env.GetLibrary("TinyPainter")
TinyHierTreeLib = Env.GetLibrary("TinyHierTree")
TinyTabLib = Env.GetLibrary("TinyTab")

scriptBox = document.getElementById("script-box")

//-----------------------------------------------------------------------
// extend the tree in TinyHierTreeLib for application related logic
//-----------------------------------------------------------------------

TinyHierTree = TinyHierTreeLib.GetObjectDef("Tree")
let NodeColors = TinyHierTreeLib.GetObjectDef("NodeColors")

let NodeFlags = JEEP.Utils.MakeFlags("", "NONE, MEMBER, VIRTUAL, ABSTRACT, IMPLEMENT")
NodeFlags.NONE = 0;

AppHierTree = Env.CreateClassDef("AppHierTree", {
	EXTENDS: [TinyHierTree],
	PROTECTED: {
		GetNodeColors__virtual: function(args){},
```

```
	},
	STATIC: {
		GetLegend: function(){},
		getColors__private: function(flag){},
	},
});

//---------------------------------------------------------------------
// extend the tab in TinyTabLib for application related logic
//---------------------------------------------------------------------

TinyTab = TinyTabLib.GetObjectDef("Tab")
TabPanel = TinyTabLib.GetObjectDef("Panel")

AppTab = Env.CreateClassDef("AppTab", {
	EXTENDS: [TinyTab],
	CONSTRUCTOR: function(){
		this.listClickHandler = this.listClickHandlerFunc.bind(this);
	},
	PUBLIC: {
		AddTab: function(id, infoArr){},
		AddUserScript: function(scr){},
		UpdateUserScript: function(scr){},
		RemoveUserScript: function(){},
		ResetCurrSel: function(){},
	},
	PROTECTED: {
		UpdateTab__virtual: function(tab){}.
	},
	PRIVATE: {
		addTab: function(id, infoArr){},
		// more
	},
})

// use the extended classes

legend = AppHierTree.STATIC.GetLegend();
// some dom node settings
document.body.appendChild(legend);

tab = AppTab.New()
tab.SetSize("100px", "170px")
tabelem = document.getElementById("tab")
tabelem.appendChild(tab.getDomElement())

tab.AddTab("Basic", [
	{name: "(SISL) single inheritance single level", code: "$Base[Derived]"},
	// more
])

// more tabs

tab.ActivatePanel(0)

canvas = document.createElement("canvas");
// some dom node settings
document.getElementById("diagram").appendChild(canvas)

Painter = PainterLib.GetObjectDef("Painter")
```

```
painter = Painter.New(canvas)

tree = AppHierTree.New(canvas.offsetWidth,
    "$Top///a[MidA,MidB] MidA///v[Der] MidB[Der]")
tree.Paint(painter)

// more cde
```

8.3. Remarks

I hope the example was able to demonstrate the superiority of using Jeep to create applications of some amount of complexity over using plain JavaScript code.

- With Jeep the code is well organized with a well defined way to create and access things. It doesn't pollute the global namespace, nor does it allow the objects to be polluted easily.
- Using objects not only makes the code intuitive and readable, it also makes adding new features quite simple.
- Having access restrictions and not exposing implementation details make the code well structured and well behaved.
- The inheritance mechanism makes extending functionalities almost trivial

9. Internals

As with most software, Jeep is in a state of flux and the internals is bound to change, for better or for worse, but hopefully for the former. So it would be futile to delve into details. Therefore, this chapter just discusses things at design level, with representative code if necessary. The design should hopefully remain more stable. This chapter cover three aspects – implementation, performance and testing – but since everything is in broad strokes, it should not be lengthy. This chapter is going to be quite opinionated, since it deals with ideas rather than unbreakable rules established by nature, so I hope that you indulge me.

9.1. Implementation

Jeep ultimately produces JavaScript code. Therefore, the only possible way for the various definitions to work is to create an object or a function and add all the members to them or their prototypes. That is somewhat close to what Jeep does.

9.1.1. Definition and Instantiation

Records are simple objects with all the variables added as its properties. With extension, the variables are simply added in sequence. Structures are also like this for most part, just that they have functions also added. Since the this of a function is the object in which it exists, structures work in a straightforward way. As of not structure and record share the same implementation functions with some object specific checks, but this will be changed soon.

During instantiation, structures need the functions to be added as properties. This is bad for performance compared to adding functions to prototype. Although the latter mechanism results in the same thing as the former, because the latter runs in the engine in native code and not in JavaScript like the former, it will always have better performance. For this reason, structures will be converted to functions in the near future.

Classes are functions but behave like objects. Firstly, the new operator is explicitly disabled in order to provide consistent interface for instantiation. Secondly, the functions are not added to the prototype as you might expect due to the access restrictions and other function directives. What I mean is, due to how these things are implemented, as explained in 9.2.2, unless a class has none of these, the functions cannot be added to the prototype. This is one major pending thing to implement in next version to extract maximum performance.

Fields and groups piggyback on the structure and class definition generation in the initial stage. They take advantage of the validations one by these objects, by supplying their own rules, and then setup their own definitions as is required. Since they are pre instantiated, they don't have any instantiation related issues.

All the necessary information for instantiation, or further definition processing in case of class inheritance, is cached inside the `_jeepdef_` object in the definition. All objects have this and each definition gets its own copy. This is what is compared inside `InstanceOf`.

Namespaces simply use a common counter to generate a token which is prefixed to the object definition name before adding to the common database. The prefix is processed away to extract the names.

9.1.2. Access Privilage and Directives

Access privilege, and some of the directives such as constantness of variables and functions are implemented by defining property using a structure called sentinel. The sentinel is captured as a closure and not accessible to outside code via `_jeepdef_` or anything else. This is a major improvement over Jeep2017, where the implementation was quite ugly, unusable, contrived with and performance, and I am sort of proud about the current implementation.

The sentinel has a few instances of `RecursiveFlag` discussed in 7.4. for various aspects. It also acts as the storage area for the variables and functions instead of the instance. The general usage pattern is shown below. This is done during instantiation since this is instance specific, and has a bearing on the performance

```
let keys = Object.keys(members)
for(let k = 0; k<keys.length; k++){
	let ky = keys[k]
	Object.defineProperty(inst, ky, {
		configurable: false,
		enumerable: true,
		get: function(){
			if(!sentinel.accessFlag.isSet())
				impl.abort("restricted-access", {})
			return sentinel.obj[ky]
		},
		set: function(v){
			if(!sentinel.accessFlag.isSet())
				impl.abort("restricted-access", {})
			sentinel.obj[ky] = v
		},
	}
});
```

The actual logic tested is much more involved, with more flags and conditions coming into play. There is one `internalAccess` that is used to circumvent the non configurability whenever members are to be updated after the instance is setup. One crucial place where this is needed is to enable and disable polymorphism. It will be discussed in 9.2.3. The abort function internally uses `MessageFormatter` discussed in 7.7.

Functions with directives are implemented as layered functions, where the actual function is sandwiched between some pre processing and post processing. In Jeep2017, each directive generated its own layered function, and combination of directives resulted in onion like nested layering which had bad performance. In the current implementation, it has been linearized. As a consequence, there is only one layered function and always it checks for all directives. This might look bad but these are just a series of if conditions, with simple bitwise operations, and simple function calls to sentinel flags, so the overhead that very much exists should not bother in practice and affect the quality of performance. The function looks some what like this.

```
let keys = Object.keys(memFuncs)
for(let k = 0; k<keys.length; k++){
	let ky = keys[k]
	let realf = memFuncs[keys[k]].item;
	inst[ky] = function(){
```

```
                if(argnum && (devode || (prodmode && pmc_argnum)){
                        if(arguments.length != realf.length)
                                impl.abort("argnum-fail", {})
                }
                // other abortable validations
                // some non abortable preprocessing
                let ex = null;
                let r = null;
                try {r = realf.apply(realf.thisObj || this, arguments)}
                catch(e){ex = e;}
                // some post processing
                if(ex) throw ex;
                return r;
        }
}
```

Even this has to be added to the instance because the function does need to use sentinel for some directives such as constantness. So this is tied to the access privilege implementation. Remember, there is also access privilege for static members.

Due to all this, the functions cannot be simply added to prototype for optimization. It needs careful analysis and is deferred to the next version.

9.1.3. Polymorphism

Due to inherent nature of JavaScript where object properties are just a key-value pair, polymorphism happens automatically. When you take a base class definition and merge the derived class with it, functions with same names automatically override the previous version that exist. For non virtual functions and all variables this is deemed wrong and appropriate error messages are issued. Yet, polymorphism s not so straightforward due to two reasons.

The first reason is semantic correctness. Polymorphism must be disabled inside constructors and destructors, so the functions must be first marked somewhere, allow the construction to happen, and then replace them in the instance. Similar thing needs to be done for destructors.

The second reason is finding the correct functions. The found functions will have to be marked as described above, but finding the correct ones is the first step. This reason exists partly due to how inheritance is implemented.

As stated, polymorphism must be disabled inside constructor. What this basically means is that inside the constructor of every base, if a polymorphic function is called, it must end up calling the class' own implementation or the one it inherited from its base, and not invoke the one in a distant derived class. So, when derived class is generated, the virtual functions at higher level in the hierarchy are used to merge. When polymorphism must be enabled, the ones at the lowest levels must be chosen. Remember, a derived class might not always implement all the virtual function of its higher bases and will simply inherit come of them. So, the entire hierarchy has to be searched to find two sets of virtual functions, one to enable and one to disable.

Accessing the entire hierarchy is needed for many validations, and also some non validating aspects like getting correct polymorphic behavior. So Jeep flattens the hierarchy tree to get better performance since array looping is always faster and simpler than recursive access.

9.1.4. Managed Instance

Jeep maintains a stack inside the framework that acts as the scope stack that the language lacks.

The stack is setup by every layered function with the managed directive in its pre processing by marking the beginning of the scope. During instantiation, it is validated that the scope has been marked and instance is pushed to the stack if everything is as expected. Every managed layered function calls `unwind` function of the stack in its post processing. This function access all the instances in LIFO order and calls their destructors.

Unwinding on exception is basically same as normal unwinding, except that the caught exception is rethrown.

9.1.5. Wrapper Class

A wrapper class is the most complex implementation detail because it has to deal with all the things discussed so far, in the process of having to implement its core feature – renaming clashing names in the hierarchy. Wrappers have by far the simplest syntax and interface and by far the most complex implementation.

Renaming is not as simple as it appears, for there exists a problem that must be solved along the way. I call this the FGK problem: if a function (or a variable) F is renamed to G and there exists a non renamed function K that uses F, the renaming must not break the code. As you can see, the renaming and the FGK problem are in direct opposition to each other and both must be solved. The only way to solve this is via composition, and a lot happens behind the scenes to get things to work correctly.

Suppose class B is wrapped by class W which is extended by class D. The class W first makes a copy of the class B and renames F to G. This way when D inherits from W, it gets G instead of F. Note that W doesn't inherit from B as that would render the entire process futile. Although this distinction is paper thin in the context of JavaScript and Jeep, it is important to maintain it.

Now, inside the constructor, W instantiates B and sets up function routing such that when d, the instance of D, calls G, the routing ultimately invokes F, and if K present in b, the instance of B, calls F it invokes F. The same thing is extended to polymorphic functions. When b calls a virtual function V, the routing invokes V of d if one exists. There is more to setting up polymorphic functions which is intricate but quite necessary. Further, the constructors and destructors must also be wrapped. All this is done while keeping access restrictions and function directives intact.

To give a more visual idea, when wrapped, an instance is split into two at the wrapping boundary. The top half until the wrapper access the members by their original names and everything in the bottom half access the members by their renamed names. The top half is the instance of the wrapped base class. The instance of the derived class will have to be attached to this via the above described routing. This happens at every wrapping boundary.

Wrapping must happen per instance. This is why it has severe performance issues, both during instantiation and usage. The latter is due to the routing mechanism that basically calls `apply` on the original functions. Again, unless you intend to instantiate thousands of instances and use them in performance critical parts of the application, or when you have just one instance but call its functions hundreds of times per second, wrapping shouldn't pose any performance problems.

9.2. Performance

Software performance is one of the most elusive things in engineering that most people mistake to have captured. I would like to think of it as a non deterministic entity. If performance is all about quantifying the speed of the code, then I stand by what I said. However, if it also includes resource management, then there is some hope but only for language like C++. For languages like JavaScript it is as hopeless as anything.

As with most things in our young industry that deal with management of software, a lot of churn has happened in this area. However not all churn is beneficial (The last time people tried churning the ocean in the hopes of getting the nectar of immortality, they first got deadly poison that threatened the world, which was consumed by a man, actually assigned with the destruction of the world, in order to save it, which made his wife choke him in order to save him. This story is full of ironies). There have been several tools and techniques devised but I find them all to be quite futile. They are looking for the wrong thing in the wrong place.

For languages like C++, performance depends on the compiler used, and depending on it, the way the source code is written. Further, with the compiler, it depends on the optimization settings used. We already have thee variables. At a deeper level, it depends on the OS on which the application runs, the current state of system resources, and ultimately the CPU that runs all the code. Beyond that, it depends on the hardware as a whole, which includes busses and whatnot. The much ridiculed phrase, “but it runs on my machine”, has a cognate wrt performance “but it is fast on my machine” and it is an inconvenient truth for software engineers to accept. I have made my peace with it a long time back.

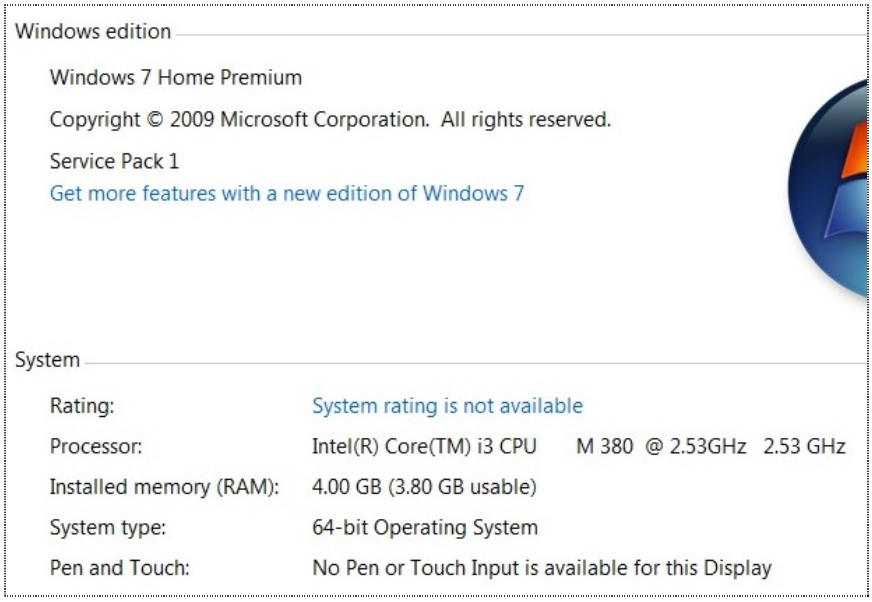
For languages like JavaScript, there are two additional variables introduced that muddles the matter further. Firstly, code runs in a restricted environment. Unlike native C++ applications, a JavaScript application doesn’t have the luxury of availing all the resources of a process. Modern browsers seem to be doing this in a limited way, but it is still a shared space. Unlike native applications where the environment, which constitutes the OS, the drivers etc remain fairly stable for long periods before updates, and bugs due to updates, browsers are rolled out every week. So measuring performance in such an unstable (for our purposes) environment is not sensible or meaningful. The analogue of the system resource state is the browser resource state, and it is an additive property that worsens matters.

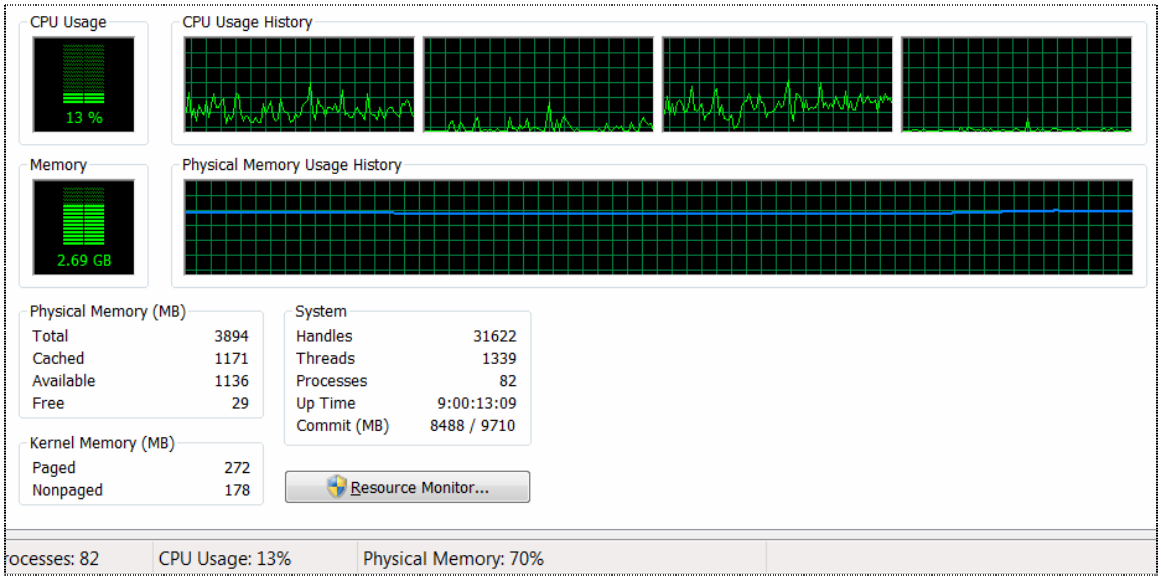
Then, the nature of JavaScript adds to the complexity. The whole garbage collector thing just messes up performance measurement. The collection of garbage is basically left to the whims and fancies of the environment, mostly the browser, and basically the engine, and there is no way to control it from the outside. Tweaking the browser to get an interface to control this is foolish because that would not be a real world scenario; it would be a sandbox inside a laboratory.

With things being this way, measuring performance is just a waste of time. What we can do instead is describe performance qualitatively, or in abstract terms and not indulge in exact quantification by producing charts and tables. Had I developed Jeep as a PhD project I would have been forced to engage in this pointless activity, but since I am not I am not going to Amusingly, for Jeep2017, despite saying something in similar vein, I did engage in the activity. But now, I will do it only partially to show how futile it is.

9.2.1. The Setup

The test was done on my personal laptop with these settings and this system state.





9.2.2. The Tests

I tested the performance with three simple scenarios before abandoning. I was always going to abandon, just that I had to show some numbers to convince you as to why abandoning is justified.

The first test was instantiation a record using InitNew in both modes. This was done to contrast it with using plain objects.

The second test was reading a value in plain object, and a variety of Jeep class instances with access restrictions and constant directives.

The third test was writing these values in the instances of the same classes used to test read.

9.2.2. The Results

The tests were run on both Firefox 60.0.2 (64 bit) and Chrome 67.0.3396.99 (Official Build) (64-bit) and the results were comparable, in terms of absurdity that is.

The columns indicate the iterations the tests was put through, and the roes indicate the time it took in milliseconds. The most striking thing is the second row in the second table. Based on this, it is slower to read value from an instance of a class that has no directives, meaning direct invocation of member functions, than one that uses layered functions due to directives. I rest my case.

record initnew	1	10	100	1000	10000
plain object	0.000	0.000	1.000	1.000	6.000
development-mode	1.000	1.000	1.000	8.000	29.000
production-mode	0.000	0.000	1.000	4.000	28.000

class value read	1	10	100	1000	10000
Non Jeep	1.000	0.000	0.000	0.000	7.000
Jeep non constant (dev)	1.000	2.000	0.000	11.000	32.000
Jeep non constant (prod)	0.000	0.000	0.000	0.000	24.000
Jeep constant (dev)	0.000	1.000	1.000	6.000	32.000
Jeep constant (prod)	0.000	0.000	0.000	3.000	20.000
Jeep private (dev)	0.000	0.000	0.000	4.000	33.000
Jeep private (prod)	0.000	0.000	1.000	2.000	25.000
Jeep private constant (dev)	0.000	0.000	1.000	4.000	37.000
Jeep private constant (prod)	0.000	1.000	1.000	2.000	14.000

9.2.4. The Conclusion

The only guaranteed and non absurd performance insights about Jeep are these:

- In development mode, processing classes in large hierarchy is always slower than processing classes with small hierarchy.
- Production mode will always be faster than development mode due to the latter completely skipping syntax and semantic validation. Refer to the end of 9.3. to see the testing report that shows some stats. As for production mode with no runtime validation, it's a tricky thing to achieve as I will explain in 9.2.
- Instantiation will always be slower in development mode, particularly InitNew.
- Functions with more directives will always be slower than functions with fewer directives.

- Having access restriction or robustness related function directives will always be slower than not having them because, as explained in 9.2.2. these must be set up during instantiation. Further, the constantness related directives will have to test all affected parties for mutation in every call. Even then, this must not be noticeable unless you instantiate thousands of instances of such objects in performance critical portion of the application.
- Instantiating classes which have a wrapped class as base is always slower than classes with non wrapped bases. The general performance will also be always slower, probably significantly, compared to classes with non bases.

9.3. Testing

As you are aware, software testing is a touchy subject for many engineers. Most of us 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. It is a C++ framework with an accompanying a 200 page document. Both are languishing in a folder since February 2017 because I think they need more refactoring an also since 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.

9.3.1. Test Cases

Testing Jeep can be divided into two parts – testing the compilation process and testing the implemented behavior. Testing compilation involves two things – syntax validation and semantic validation. syntax validation would be identifying an unknown directive used, and semantic validation would be identifying an inappropriate directive being used, for example a field function using `abstract` directive. Testing behavior involves two things – the core behavior and runtime validation. The core behavior would be invocation of correct virtual function and runtime validation would be refusing to instantiate a class with unimplemented abstract functions.

For the sake of completeness, the `Utils` must also be tested.

All these must be done for both environments. The production mode skips all compilation validations, but it must be tested that it skips. So, almost all test cases are run twice, once for development mode and once for production mode. The remaining cases are exclusively for production mode. They test the PMC only. since PMC is irrelevant in development mode and validation is always done, only production mode needs to test this. These tests could have been done in both modes also, but it didn't seem right to do futile tests and inflate the case count.

The tests cover both passing and failing scenarios. Testing any functionality that throws an exception is a failing scenario, but as with everything here are exceptions (ha!) to this.

9.3.2. The Setup

The cases are contained as a key value pair, the key being the test name and the value being the function to run. A small general purpose testing framework is bundled with Jeep using which it is tested. The basic idea is to run the test function and let them generate a series of outputs and compare them with a predetermined set of expected output. Tests pass if both match.

The test functions look like this.

```
"val-equal-native": function(env, cout, info){
	info.desc = "tests ValEqual with non JEEP objects"
	info.exp = ["ValEqual: false"];
	info.aspects = "core";

	cout("ValEqual:", JEEP.ValEqual(10, 10) ? "true":"false")
},
```

Every test function is given the environment, an output logging function and an info object, which must be filled with the test details. The `desc` must contain the test description. The name is only to identify the test loosely, and make unique test objects, so it can be a bit cryptic. It's the description that really explains what the test is all about. This will shown in the report, so it must be reasonably descriptive while not being verbose. The `exp` is the expected result. It is an array of strings. The `aspects` is a comma separated string intended to indicate what aspects of the framework are being tested in the function. The aspects will be part of the report. They help identify the quality of the tests performed and also the coverage of features. The `cout` is setup such that the text gets added to the generated list. After the function exits both lists are compared and results are logged to the report.

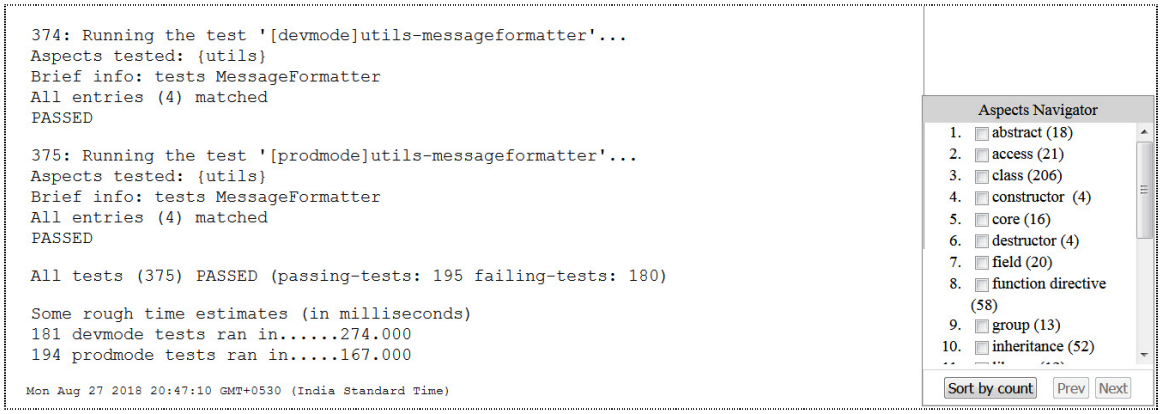
For some tests, particularly the compilation validation, there needs to be two expected results, one for each mode. The environment object is used to set it up.

```
"struct-access": function(env, cout, info){
	info.desc = "tests the access restriction of structure"
	if(env.IsDevMode())
		info.exp = [
		"JEEP: Attempt to write private variable 'Test.value' detected.",
		"JEEP: Attempt to access private function 'Test.change' detected.",
		"Test value = 10"
		];
	else
		info.exp = ["Test value = -2"]
	info.aspects = "struct, access";

	let Struct = env.CreateStructDef("Test", {
		CONSTRUCTOR: function(x){this.change(x);},
		print: function(){cout(this.$name, "value =", this.value)},
		value__private: 0,
		change__private: function(v){this.value = v},
	});
	let s = Struct.New(10);
	try{s.value = -1}catch(e){cout(e)}
	try{s.change(-2)}catch(e){cout(e)}
	s.print();
},
```

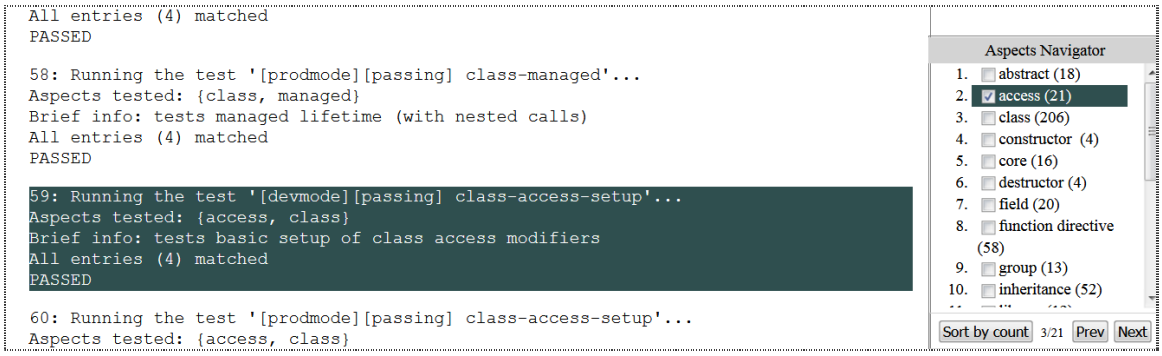
9.3.3. The Report

The report lists all the test results, whether pass or fail, along with the description and aspects. It goes without saying that the test case count is bound to change – new tests for features could be added, existing cases could be merged or split or discarded – so any count shown here should be taken only as a representative number.



The test names are prefixed with the mode and the nature of the test. This is useful in general, but in particular when a test fails in one of the four combinations. The aspects are shown in two places: in the test result and a separate panel.

The panel allows some user interaction and is intended to help navigate tests that incorporate these aspects. Each aspect shows a number indicating how many of it were tested. The aspects can be sorted based on names or occurrence count. Clicking on an item highlights all the tests testing that aspect and allows simple navigation within that set using the buttons. Clicking also checks the checkbox at the beginning. You can browse combination of aspects by checking multiple boxes and navigating the resulting set. If such combinations are invalid the buttons are disabled. When navigation is valid, the state of navigation is shown between the buttons in the form n/m, where n is the current index, and m is the size of the set of ests that have the selected aspects.



At the end, the report shows some statistics. It breaks up the test into passing and failing ones, and also the development mode ones and production mode ones. For the latter split, it also shows running times. Remember, the exact same tests were run in both modes. As you can see, the production mode tests are significantly faster than development mode ones.

Errors will be indicated by red text for the test result, and all failed tests will be listed at the end as hyperlinks within the report. All failed tests can be navigated using a small panel attached to each failed test listing.

```
Total tests: 361 (passing-tests: 185 failing-tests: 176) Passed: 357 Failed: 4

Some rough time estimates (in milliseconds)
174 devmode tests ran in......290.000
187 prodmode tests ran in.....132.000
```

1. [devmode][passing] namespace-names
2. [prodmode][passing] namespace-names
3. [devmode][passing] namespace-object-names
4. [prodmode][passing] namespace-object-names

```
133: Running the test '[devmode][passing] namespace-names'...
Aspects tested: {namespace}
Brief info: tests the namsepace structure

**  expected : Rec Struct Class
**  generated: .ns0.Rec .ns0.Struct Class

The test '[devmode][passing] namespace-names' FAILED
```

1/4 prev next listing

10. Conclusion

10.1. A Review

If you have read the entire document, you would have seen how extensive and complex is Jeep. Had it been created at a megacorp, Jeep would have been be unabashedly 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 needed a document of this length.

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 "object orientation" 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 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 programming style rather than being called so just because functions are passed around.

- For code that creates a lot of objects intended as entries for arrays or key values for 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 Classes. 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. Further, it introduces the concept of instance lifetime and destructors, and offers mechanisms of scope creation for automatic invocation of the destructors that help manage resource efficiently and automatically.

- Jeep offers three levels of privilege for accessing class members – public, protected and private. The programmer can pick the right privilege level depending on the necessary semantics.

- 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.

- Jeep offers a solid mechanism of class inheritance and building class hierarchies. It greatly improves the single inheritance aspect that the JavaScript language has in a very crude way.

- Jeep offers multiple inheritance facility that is absent in the language. It offers simple and elegant solutions to the conceptual problems that arises with multiple inheritance.

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

- Jeep introduces the concept of working in development mode and production mode, where the former is intended for the developer and the latter for the end user. Consequently, the former provides maximum assistance in debugging. and as a result is slower, which is acceptable. In contrast the latter is made as fast as possible, retaining only elementary debugging information.

- Jeep allows the clients to work in development mode and production mode on similar lines for better software engineering.

- Jeep introduces the concept of library, and offers a rather unique and efficient way of building them. Libraries organize objects and allows access to them elegantly, and relieves the client of having to juggle a host of objects.

- 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' if you stick solely to DOM event handlers for simple front end applications. For anything other than that, it is an emphatic 'yes'. Beyond web front end, there are two more scenarios where Jeep will assist greatly.

One is server side development which is increasingly being done in JavaScript with software like node.js. and the other is desktop development done with JavaScript using software like Electron. On both counts I am sure there will be more competing software coming up in the near future. While it is true that you can link to native C++ code from these software to get good performance, thus relegating 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 and deployment that can happen with JavaScript compared to traditional languages like C++, and the freedom the language offers over other languages makes JavaScript very attractive for serious development. The only problem with the language was its fragility but Jeep resolves most of them.

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, some questions are pertinent and I try to answer them in the next section.

12.2. Some Questions and Answers

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

Indeed, JavaScript is improving by the day. But then again, there is hardly anything worthy done at structural level, except syntactic sugars. Jeep offers much more, as was explained rather extensively in this document. So, until the language catches up, Jeep is needed and 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 megacorps). 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 aspects of the language and continue to use a robust version of JavaScript. Why waste all the invested time?

3. Why isn't there function overloading which is quite a prominent C++ feature?

It's a two fold answer. First, it is not there because I don't like it. I don't like it because there are not many useful scenarios for it. The only one I can think of is having a function like `Log` instead of `LogText`, `LogNum` etc, that allows simpler usage. Second, this is a built in feature of JavaScript and works in most cases with the same code. For the cases that need special code, thus requiring an overloaded function, it seemed a lot of work to implement something that I don't like and don't find useful. Further, such a special case can be easily handled within the function itself using plain JavaScript techniques, so it seemed that much more futile to have this non feature.

4. Does Jeep work in strict mode?

To be honest, I don't know. Firstly, it seems that strict mode is not implemented consistently across browsers. Then, I am a novice in JavaScript and have not used this mode in practice. For both these reasons, I haven't even attempted to see if it runs in strict mode.

5. Jeep seems interesting but I wouldn't use it. Is that alright?

Well, first of all I appreciate that you find it interesting. As for not using it, it is of course alright if you don't want to. I only hope you have read the entire document before making this decision. If you haven't, then you must. Jeep is quite unlike JavaScript, and it needs some time to be appreciated. Have you tried building something with it? There is no need to rush to any decisions. However, if you have read the document completely, and tried Jeep, and still feel so, then it would be beneficial if you are a bit more clear as to why you don't want to use it.

6. What did you hope to achieve with Jeep?

To be honest, I just wanted to build my web applications the way I built my desktop ones, but this ended up being fun. I have not written a serious compiler, and doing this was very close to it. Apart from that, once I saw what was possible, I secretly hoped to shame the standards committee and the engine developers to incorporate at least some of these features natively. I mean, if these can be done with Jeep, they can surely be done with native code written in C++.

12.3. What's Next?

For most part 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 issues etc.

However, there are a few things that are interesting enough to be attempted, but still not important to have in Jeep. Some are such features that are garnishes which simply exist to increase the price tags. Others are ambitious hacks to make Jeep more mainstream.

- Making just one function argument constant instead of making all of them constant.
- Adding more directives to constructors, especially argument type validation and constantness.
- Mechanism to convert plain JavaScript class to Jeep class to use them in hierarchies.
- Hacking an open source engine to directly process Jeep so that Jeep code becomes native to that engine, thereby reducing the current Jeep implementation to a shim!
- Generating meta content for hierarchies to use in visualizing tools like the one demonstrated.

Beyond this tentative and incomplete list, I am sure more will come up during *enjeepification*.

12.4. Some Last Words

I began with a personal story, so it is fitting that I end with one.

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 (Amid the third iteration, my mother underwent a surgery and I spent a lot of time at the hospital where I didn't like to take my laptop. So I tried coding on my Galaxy Tab by downloading firefox and a code editor. I couldn't last five minutes. May be smart TV would be different). 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 I would just say why not? For a less facetious answer, you would have to wait till I publish them.

ಶುಭಂ

Reference

R.1. Main Object

The framework is contained within JEEP object which has the following properties

```
JEEP = {
       InitFramework: function(){},
       SetStdErr: function(logger){},
       CreateEnvironment: function(info){},
       ObjectDefExists: function(name){},
       GetObjectDef: function(name){},
       Typedef: function(newtype, existing){},
       RegisterLibrary: function(name, initFunc){},
       Equal: function(a, b){},
       ValEqual: function(a, b){},
       Utils: {},
       impl: {},
}
```

The `impl` object contains the implementation details and is supposed to be not accessed directly.

The `Utils` object contains some utilities which are exposed to the client. These utilities are used in Jeep's own implementation, so it is sort of pre tested in production mode apart from being formally tested along with Jeep itself.

R.1.1. InitFramework

Description

This function sets up the framework and is supposed to be called before using the framework, failing which the framework will fail to work.

Input Arguments

The function takes no arguments.

Return Value

The function returns nothing..

R.1.2. SetStdErr

Description

This function is used to change the logging function used by the framework to issue errors and warnings. By default, the framework logs to the console.

Input Arguments

The function takes one argument which is supposed to be a function taking one argument and returning undefined. This function is supposed to do the actual logging of the given input.

Return Value

The function returns the existing logging function. Using the return value, a caller can set and reset the logging functions.

Example

```
JEEP.SetStdErr(function (t){
       let p = document.createElement("p");
       p.textContent = t;
       document.getElementById("log-div").appendChild(p)
})
```

R.1.3. CreateEnvironment

Description

This is the most important function after the `InitFramework`. It creates an environment inside which all Jeep objects are created. The created object is immutable.

Input Arguments

The function takes one object which has these two properties:`{mode, client, flags}`

- `mode` is a string that takes one of these values: `development-mode, production-mode.`
- `client` is a string that takes one of these values: `jeep-aware, jeep-agnostic.`
- `flags` is a string that takes some flags that affect validation
    - *`trap-disabled-virtual-call:`* This is valid in development mode only and generates runtime error if virtual functions are invoked from a call chain started by the constructor and destructor of a class.

The mode and client ate mandatory but flags is optional.

Return Value

The function returns an object with these properties. The object layout is same for all info combinations; only the behavior is affected. This will be explained in the R.2.

This function throws exception if

- argument contains properties other than the ones mentioned
- mode and client are not given
- mode and client have invalid values
- if flags contains values other than the ones mentioned

Remarks

The mode property has significant impact on how Jeep functions behave. In development mode, all validation is done. Validations apply to three aspects – syntax, semantics and runtime behavior. Consequentially this has a bearing on the performance. In contrast, in production mode the syntax and semantics validation is skipped completely. Most runtime validations can be retained by providing associated flags during object definition creation. Since this is an object specific setting, more about this will be explained in respective object sections.

The client property is intended to be used by clients such as libraries. It doesn't cause any behavioral changes by itself; but rather, it provides the clients a hint based on which they may affect the changes. This is intended to control runtime validation in production mode via the flags mentioned above.

R.1.4. ObjectDefExists

Description

This function tells if a definition is registered with `RegisterX` functions of the environment object.

Input Arguments

This function takes a string which is the name of the object. This name is supposed to be the same as the one used while registering.

Return Value

This function returns a Boolean value depending upon the existence of the definition.

Remarks

This function exists in the main object instead of the environment because, unlike creation of objects, checking for definition has no behavioral changes imposed by environment.

This function is useful as a replacement to `GetObjectDef` which throws exception if the definition was not registered since testing the Boolean is much better for performance than try-catch block and amounts to lesser code.

R.1.5. GetObjectDef

Description

This function retrieves the definition of objects that were registered via the `RegisterX` functions in the environment object.

Input Arguments

This function takes a string which is the name of the object. This name is supposed to be the same as the one used while registering.

Return Value

This function returns the object definition using which objects can be instantiated.

The function throws an exception

- if the given name was not found in the object database.

Remarks

This function exists in the main object instead of the environment because, unlike creation of objects, retrieval has no behavioral changes imposed by environment.

R.1.6. Typedef

Description

This function creates a new type and registers it. This function will be explained in the context of namespace and function validation since it is intended to be used there.

Input Arguments

This function takes two string arguments. The first is the name for the new type being defined and the second is the name of the existing type. to be used as a reference

Return Value

This function returns nothing.

This function throws an exception

- if the names contains invalid characters
- if the reference type is not found
- if the intended new type name already exists,

Remarks

This function exists in the main object instead of the environment because defining types has no behavioral changes imposed by environment.

Example

```
DemoEnv.RegisterRecordDef("Record", {/*details*/})
JEEP.Typedef("MyRecord", "Record");
let Record = JEEP.GetObjectDef("MyRecord");
let r = Record.New();
// use record
```

R.1.7. RegisterLibrary

This is explained in R.9.

R.1.8. Equal

Description

This function tests equality of two objects.

Input Arguments

This function takes two arguments which are the objects that must be tested for equality.

Return Value

This function returns a Boolean value depending on the equality of the input arguments.

Remarks

This function expects Jeep object instances only. So, even if two numerical constants are given as input, the function returns `false`.

Two conditions must be fulfilled for the equality test – being of same type and having same value. This function can actually be decomposed into two other functions, both of which exist in the framework. One is `InstanceOf` that every object definition has, and the other is `ValEqual` described next. However, this function exists to simplify the operation.

R.1.9. ValEqual

Description

This function tests if two objects have same values.

Input Arguments

This function takes two arguments which are the objects that must be tested for equality.

Return Value

This function returns a Boolean value depending on the equality of the input arguments.

Remarks

This function expects Jeep object instances only. So, even if two numerical constants are given as input, the function returns `false`.

This functions tests equality of values only, so a class and a record having same variables names with same values are considered equal.

The `Equal` function uses this function internally in principle.

R.2. Environment

Before objects can be defined, retrieved and used, a suitable environment must be created with the `CreateEnvironment` function explained in R.1.3. The created environment object looks like this.

```
Environment = {
	IsDevMode: function(){},
	IsClientJeepAware: function(){},
	CreateRecordDef: function(name, def){},
	RegisterRecordDef: function(name, def){},
	CreateStructDef: function(name, def){},
	RegisterStructDef: function(name, def){},
	CreateClassDef: function(name, def){},
	RegisterClassDef: function(name, def){},
	CreateField: function(where, def){},
	RegisterField: function(name, def){},
	CreateGroup: function(where, name, def){},
	RegisterGroup: function(name, def){},
	CreateNamespace: function(){},
	GetLibrary: function(name){},
}
```

R.2.1. IsDevMode

Description

This function is supposed to be used to know about the mode of the environment.

Input Arguments

This function takes no arguments.

Return Value

This function returns a Boolean value set to true or false depending on whether the environment is in development mode or production mode.

R.2.2. IsClientJeepAware

Description

This function is supposed to be used by the libraries to adjust code generaton based on the type of the intended client.

Input Arguments

This function takes no arguments.

Return Value

This function returns a Boolean value set to true or false depending on the type of client.

R.2.3. CreateX

Description

These functions, except `CreateNamespace`, create the object definition intended to be used within a local scope beyond which they are inaccessible, unless they are exposed outside the scope by some mechanism.

Input Arguments

The functions take two arguments. The first is a string for the name of the object. The second is a definition object which is different for different objects. It will be explained while explaining the associated object.

Return Value

This function returns the created object definition.

This function throws exceptions

- if the name contains invalid characters
- if syntax and semantics validation fail (which are object dependent)

Remarks

These functions create the exact same object as the corresponding registering functions. The only difference is the default scope of accessibility.

Since this is a local scope object, the name has no consequence beyond identifying the object and its instances.

R.2.4. RegisterX

Description

These functions create the object definition intended to be used in a global scope. The objects are stored in a database and can be retrieved using the `GetObjectDef` explained in R.1.5.

Input Arguments

The functions take two arguments. The first is a string for the name of the object. The second is a definition object which is different for different objects. It will be explained while explaining the associated object.

Return Value

This function returns no result.

This function throws exceptions

- if the name contains invalid characters
- if the name is already registered
- if syntax and semantics validation fail (which are object dependent)

Remarks

These function creates the exact same object as the corresponding creating functions. The only difference is the default scope of accessibility.

The name given here is what should be used to retrieve the definition with `GetObjectDef`.

R.2.5. CreateNamespace

Description

This function creates a namespace. Namespaces are explained in R.3.

Input Arguments

This function takes no arguments.

Return Value

This function returns a namespace object.

R.2.6. GetLibrary

This is explained in R.9

R.3. Record

A record is supposed to be pure data with only member variables and not functions. However, since functions are first class objects in JavaScript, the member variables can be functions as well. However, such functions are treated as pure data and are not put through special processing as is done with other objects. Therefore, while you can hack a structure using a record, you will face severe usage inconveniences and performance penalties

R.3.1. Definition

The `CreateRecordDef` and `RegisterRecordDef` functions generate the definition. The first function returns the definition as its result which is accessible only in local scope unless it is exposed to outer scope via some mechanism. The second function adds the definition to a

common database which makes it accessible in global scope. It doesn't return any result. The function exists in two places, one in the environment object and one in a namespace object. The one in the environment adds the definition to the database directly while the one in the namespace disambiguates it before adding to the database. The registered definition can be retrieved via `GetObjectDef` function and its existence can be checked via `ObjectDefExist` function, both present the JEEP object.

Both generation functions have the same syntax of the following form.

```
CreateRecordDef: function(name, def){},
RegisterRecordDef: function(name, def){},
```

The first argument `name` is the name of the record. For the creating function, this serves only in identifying the instances. However, this is crucial for the registering function. In addition to help identify the instances, it also acts as a key value for the definition object added to the database. The `GetObjectDef` and `ObjectDefExists` functions are supposed to be passed this name.

The second argument `def` is an object describing the record definition. The declaration has the following general layout.

```
{
	EXTENDS: [<record names>|<record definiton>],
	<variable name>: <value>,
}
```

The values assigned to the variables are considered default values. These can be selectively changed during instantiation.

The special property `EXTENDS` is optional. This is the only special property valid for records. It is an array that can contain strings and objects. The strings are supposed to be the names of registered records, and the objects the record definitions, which may be either created or retrieved. When this property exists, the record will automatically get the variables from the mentioned records.

These functions throw exception

- if name is invalid
- if the definition contains properties not intended for records
- if EXTENDS contains non string and non record definitions
- if EXTENDS contains names of unregistered records
- if extended records cause name clashing

In addition, the registering function throws exception

- if name is already registered

The generated definition has the following layout.

```
Record = {
	New: function(){},
	InitNew: function(ob){},
	InstanceOf: function(inst){},
```

```
        _jeepdef_: {}
}
```

The `_jeepdef_` object contains the implementation details and is not supposed to be touched.

The `InstanceOf` function takes one argument. It returns a Boolean value depending on whether the input is an instance of the record.

The `New` and `InitNew` functions are related to instantiation and are explained next.

R.3.2. Instantiation

A record can be instantiated using two functions `New` and `InitNew` both of which are present in the definition object. Both functions return a new instance of the record.

The `New` function takes no arguments and creates an instance with the default values. For extended records, the default values are the ones mentioned in the source record declaration.

This function throws exception

- if invoked with arguments

The `InitNew` function takes a single object as its argument which is supposed to contain values to be assigned to the variables. The object is supposed to be a subset of the variables present in the definition. These variables include the ones merged due to extension.

This function throws exception

- if invoked with zero arguments
- if invoked with more than one argument
- if invoked with an object that is not subset of the variables

The created instance will have the following layout.

```
inst = {
        <varable name>: <value>
        $name: “”,
        $def: {}
        _jeepdef_: {}
}
```

The variables and the values are the same ones mentioned in the declaration. They include the ones merged due to extending the record. When instantiated with `InitNew`, some of them might have different values based on the input to that function.

The reflective property `$name` contains the same name that was used during generation. This helps in identifying an instance by name in application code. The reflective property `$def` is exactly the same as the generated definition. This acts as an easily accessible definition when an instance is at hand.

The `_jeepdef_` object is the same one present in the generated definition and is not supposed to be touched.

R.4. Structure

A structure is a light weight class and a more feature rich record. It is supposed to be used to define small utilities such as stack, string processor etc. It can have member variables and member functions. The functions are put through special processing to generate the definition.

R.4.1. Definition

The `CreateStructDef` and `RegisterStructDef` functions generate the definition. The first function returns the definition as its result which is accessible only in local scope unless it is exposed to outer scope via some mechanism. The second function adds the definition to a common database which makes it accessible in global scope. It doesn't return any result. The function exists in two places, one in the environment object and one in a namespace object. The one in the environment adds the definition to the database directly while the one in the namespace disambiguates it before adding to the database. The registered definition can be retrieved via `GetObjectDef` function and its existence can be checked via `ObjectDefExist` function, both present the JEEP object.

Both generation functions have the same syntax of the following form.

```
CreateStructDef: function(name, def){},
RegisterStructDef: function(name, def){},
```

The first argument `name` is the name of the record. For the creating function, this serves only in identifying the instances. However, this is crucial for the registering function. In addition to help identify the instances, it also acts as a key value for the definition object added to the database. The `GetObjectDef` and `ObjectDefExists` functions are supposed to be passed this name.

The second argument `def` is an object describing the structure definition. The declaration has the following general layout.

```
{
	CONSTRUCTOR: function(){},
	PMC: “”,
	<variable name>: <value>,
	<function name>: <function definition>,
}
```

Only variables is treated as data and member functions are supposed to work on them. Variables always have public access. Functions have public access by default but they can be made private. The values assigned to variables are considered default values. These can be selectively changed during instantiation. The special properties `CONSTRUCTOR` and `PMC` are optional. These are the only special properties valid for structures.

The member functions can have the `private` directive. This is the only directive allowed for structure member functions. This directive renders the declared function private and inaccessible to functions other than other member functions. This directive is always in effect in development mode and retained optionally in the production mode.

`CONSTRUCTOR` is the first function called upon instantiation if one is present. This function is completely user defined.

`PMC` stands for "production mode checks". It is a string that contains flags to retain some robustness features in production mode. The only valid flag for structure is `pmc-memaccess`. This flag retains the effect of the private directive in production mode.

These functions throw exception

- if name is invalid
- if the declaration contains properties not intended for structures
- if member functions have directives not intended for structures
- if PMC contains flags not intended for structures

In addition, the registering function throws exception

- if name is already registered

The generated definition has the following layout.

```
Struct = {
       New: function(){},
       InitNew: function(ob){},
       InstanceOf: function(inst){},
       _jeepdef_: {}
}
```

The `_jeepdef_` object contains the implementation details and is not supposed to be touched.

The `InstanceOf` function takes one argument. It returns a Boolean value depending on whether the input is an instance of the record.

The `New` and `InitNew` functions are related to instantiation and are explained next.

R.4.2. Instantiation

A structure can be instantiated using two functions `New` and `InitNew` both of which are present in the definition object. Both functions return a new instance of the structure. Before the instance is returned, the constructor is invoked if present.

The `New` function creates an instance with the default values. It can take any number of arguments but only when a constructor is defined, in which case the constructor is invoked with the same arguments. The constructor sees the default values for the variables.

This function throws exception

- if invoked with arguments when no constructor is defined

The `InitNew` function takes a single object as its argument which is supposed to contain values to be assigned to the variables. The object is supposed to be a subset of the variables present in the definition. These variables include the ones merged due to extension.

This function throws exception

- if invoked with zero arguments
- if invoked with more than one argument
- if invoked with an object that is not subset of the variables

The created instance will have the following layout.

```
inst = {
       <varable name>: <value>
       <function name>: <function definition>,
       $name: “”,
       $def: {}
       _jeepdef_: {}
}
```

The variables and the values are the same ones mentioned in the declaration. When instantiated with `InitNew`, some of them might have different values based on the input to that function. However, a constructor can always change the values to something else based on the arguments it takes.

The reflective property `$name` contains the same name that was used during generation. This helps in identifying an instance by name in application code. The reflective property `$def` is exactly the same as the generated definition. This acts as an easily accessible definition when an instance is at hand.

The `_jeepdef_` object is the same one present in the generated definition and is not supposed to be touched.

R.5. Class

A class is the most complex object and has full fledged object orientated programming features. It also has more robustness features than any other object.

R.5.1. Definition

The `CreateClassDef` and `RegisterClassDef` functions generate the definition. The first function returns the definition as its result which is accessible only in local scope unless it is exposed to outer scope via some mechanism. The second function adds the definition to a common database which makes it accessible in global scope. It doesn’t return any result. The function exists in two places, one in the environment object and one in a namespace object. The one in the environment adds the definition to the database directly while the one in the namespace disambiguates it before adding to the database. The registered definition can be retrieved via `GetObjectDef` function and its existence can be checked via `ObjectDefExist` function, both present the JEEP object.

Both generation functions have the same syntax of the following form.

```
CreateClassDef: function(name, def){},
RegisterClassDef: function(name, def){},
```

The first argument `name` is the name of the class. For the creating function, this serves only in identifying the instances. However, this is crucial for the registering function. In addition to help identify the instances, it also acts as a key value for the definition object added to the database. The `GetObjectDef` and `ObjectDefExists` functions are supposed to be passed this name.

The second argument `def` is an object describing the structure definition. The declaration has the following general layout.

Since class is the most complex object, the definition is also more complex with more properties than other objects.

```
{
	EXTENDS: [<names>|<definitions>],
	CONSTRUCTOR: function(){},
	DESTRUCTOR: function(){},
	PMC: “”,
	PUBLIC: {
		<variable name>: <value>,
		<function name>: <function definition>,
	},
	PROTECTED: {
		<variable name>: <value>,
		<function name>: <function definition>,
	},
	PRIVATE: {
		<variable name>: <value>,
		<function name>: <function definition>,
	},
	STATIC: {
		<variable name>: <value>,
		<function name>: <function definition>,
	},
}
```

R.5.1.1. CONSTRUCTOR

This is the first function called upon instantiation if one is present. This function is completely user defined. Its behavior will be explained in the Instantiation section.

The only directive allowed is `managed`. Classes with this directives can only be instantiated inside a function that has the `managed` directive of any function in the call chain initiated by it. When this directive is used, the class should define a destructor. Not doing so generates compile time error.

Virtual functions are not only disabled inside the constructors, attempt to invoke them generates runtime failure.

R.5.1.2. DESTRUCTOR

This is the last function invoked before the instance is destroyed if one is present. This function is relevant only for managed classes, so only managed classes should define this. A compiler error is generated if either managed classes don’t define a destructor or one is defined for non managed

classes. This function cannot throw an exception. Doing so is a serious structural error and JEEP aborts immediately. This function cannot have any directives.

Virtual functions are not only disabled inside the destructors, attempt to invoke then generates runtime failure.

R.5.1.3. Access Privilege

The variables and functions exist in four levels of access privileges. Public members are accessible to any function. Protected members are accessible only to derived classes and member functions of the class. Private members are accessible only to member functions of the class. These three privileges need instances to access. Static members are accessible without an instance. Such members are shared by all instances.

Static members by default have public access but can me made private by using directives. Private static members are accessible only to member functions of the class. Protected static members don't exist as they make no sense and offer no benefit.

Both static and non static access privileges are always in effect in development mode and retained optionally in the production mode.

R.5.1.4. Variable Directives

Only the following directives are allowed for variables

- `const`: This renders the variable constant. No function, including member functions can modify the value once it is set. Only the constructor can set the value if needed.
- `get`: This generates a function named `get_<varname>` which returns the value.
- `set`: This generates a function named `set_<varname>` which sets the value. This directive cannot be used for constant variables since it is absurd. Attempt to do so will generate compile time error.

R.5.1.5. Function Directives

Only the following directives are allowed for functions

- `const`: This renders the function constant, which means the function cannot change any member variable values. Attempt to change values generates run time error. This directive is always effective in development mode and can be selectively retained in production mode.
- `argnum`: This validates that the function is invoked with exactly the same number of arguments as is declared in the function definition. Attempt to invoke with different number of arguments generates run time error. This directive is always effective in development mode and can be selectively retained in production mode.
- `argnumvar`: This validates that the function is invoked with at least the same number of arguments as is declared in the function definition. Attempt to invoke with fewer number of arguments generates run time error. This directive is always effective in development mode and can be selectively retained in production mode.
- `argconst`: This renders the arguments constant. Attempt to change the values generates run time error. This directive is always effective in development mode and can be selectively retained in production mode.

- `argtype`: This validates that the function is invoked with exactly the same number of arguments og exactly the same types as is declared in the function definition. Attempt to invoke with different number of arguments or different types generates run time error. This directive is always effective in development mode and can be selectively retained in production mode. The types are indicated via directive syntax.
    - `v__number`: indicates the argument v is of `number` type
    - `v__string`: indicates the argument v is of `string` type
    - `v__array`: indicates the argument v is of `array` type
    - `v__object`: indicates the argument v is of `object` type
    - `v__myobj`: indicates the argument v is of a registered type `myobj`
- `managed`: This provides the function with a managed scope inside which classes with managed constructors can be instantiated. This is a core feature and is always on irrespective of environment mode.
- `virtual`: This renders the function polymorphic. This is a core feature and is always on irrespective of environment mode.
- `abstract`: This renders the function polymorphic but also enforces that the derived classes implement their own versions. A class with abstract functions cannot be instantiated. Attempt to do so will generate runtime error. This is a core feature and is always on irrespective of environment mode.
- `private`: This is valid only for static functions and generates compile time error if used for member functions. This renders the function private. This directive is always effective in development mode and can be selectively retained in production mode

R.5.1.6. PMC

PMC stands for "production mode checks". It is a string that contains flags to retain some robustness features in production mode. The following are the only flags valid for classes.

- `pmc-memaccess`. This flag retains the access privilege restriction in production mode. It applies to both static functions with `private` directive and non static functions defined within respective groups.
- `pmc-argconst`: This flag retains the effect of `argconst` directive.
- `pmc-argnum`: This flag retains the effect of `argnum` and `argnumvar` directives.
- `pmc-argtype`: This flag retains the effect of `argtype` directive.
- `pmc-constant-function`: This flag retains the effect of `const` function directive.
- `pmc-constant-variable`: This flag retains the effect of `const` variable directive.

R.5.1.7. EXTENDS

This is an array of strings or class definition objects. The strings are supposed to be the names of registered classes. The mentioned classes become the base classes for the class being defined. The class inherits everything from all the bases except the `PMC` property, which is class specific. Due to this there are some validations done and compiler errors issued if they fail. This happens only in development mode.

- member names cannot repeat.
- a function defined as virtual must always be defined as virtual in the class hierarchy.
- a function defined as abstract must always be defined as virtual in the class hierarchy after being implemented.
- a virtual of abstract function must have the exact same argument count and directives in the class hierarchy.

- a virtual or abstract function cannot repeat in more than one class unless they are inherited from a common base.
- the class should be declared as managed if at least one base is declared as managed.

These functions throw exception

- if name is invalid
- if the declaration contains properties not intended for structures
- if any of the above mentioned property specific validations fail

In addition, the registering function throws exception

- if name is already registered

The generated definition has the following layout.

```
Class = {
       New: function(){},
       InitNew: function(ob){},
       InstanceOf: function(inst){},
       _jeepdef_: {}
}
```

The `_jeepdef_` object contains the implementation details and is not supposed to be touched.

The `InstanceOf` function takes one argument. It returns a Boolean value depending on whether the input is an instance of the record.

The `New` and `InitNew` functions are related to instantiation and are explained next.

R.5.2. Instantiation

A class can be instantiated using two functions `New` and `InitNew` both of which are present in the definition object. The latter method of creating instance is called "initialized construction". Both functions return a new instance of the class.

Before the instance is returned, the constructor is invoked if present. If a class has base classes, the constructors are called from top most base class to the bottom most class, which is the class being instantiated. When more than one base exists, the same vertical order is maintained but applied to each class listed in the `Extends` from left to right.

The instantiation is aborted if any constructor throws an exception. Constructors can also simply return false to indicate failure, which also aborts the instantiation. Before aborting, all the corresponding destructors, if present, of the bases which constructed successfully are invoked in the opposite order of construction. The destructors are also invoked in the same order when an instantiated managed class goes out of scope. If any destructor throws an exception JEEP aborts the entire application.

The `InitNew` function takes a single object as its argument which is supposed to contain values to be assigned to the variables. The object is supposed to be a subset of the variables present in the definition. These variables include the ones merged due to extension.

When initialized construction happens, the instance gets a Boolean reflective property `$initnew`. This is a temporary property and available only to the constructors during instantiation. This is intended to be used by the constructors to decide if variables need to be initialized.

This function throws exception

- if invoked with zero arguments
- if invoked with more than one argument
- if invoked with an object that is not subset of the variables
- if a managed class is instantiated in non managed scope
- if one of the constructors throws an exception

The `New` function creates an instance with the default values. It is more flexible and can take any number of arguments. When exactly one argument is given, this function does some processing as follows. For more than one, it simply passes the arguments to the constructors.

1. It first attempts copy construction if the argument is of the same type. When copy construction happens, the instance will get a copy of the variable values from the input argument and the constructor is not called.

2. If copy construction is not feasible, the function checks if initialized construction can happen. Unlike the `InitNew` function, this function doesn't throw exception but simply skips the process if the input fails to meet the requirements. If it does meet, the constructor is invoked.

3. If both attempts fail, the function simply passes the argument to the constructor as it does when more than one arguments are given.

This function throws exception

- if a managed class is instantiated in non managed scope
- if one of the constructors throws an exception

The created instance will have the following layout.

```
inst = {
       <varable name>: <value>
       <function name>: <function definition>,
       $name: “”,
       $def: {}
       _jeepdef_: {}
}
```

The variables and the values are the same ones mentioned in the declaration. When instantiated with `InitNew`, some of them might have different values based on the input to that function. However, a constructor can always change the values to something else based on the arguments it takes.

The reflective property `$name` contains the same name that was used during generation. This helps in identifying an instance by name in application code. The reflective property `$def` is exactly the same as the generated definition. This acts as an easily accessible definition when an instance is at hand.

The `_jeepdef_` object is the same one present in the generated definition and is not supposed to be touched.

R.6. Field

A field is the exact opposite of a record in that it has only functions and no variables. It is just a named collection of functions. A field is also a stripped down class. All functions have public access. The only thing it has in common with a class is that the functions can have some of the directives that are not related to classes. So functions cannot have `virtual` and `abstract` directives. Since there are no variables, functions cannot have the `constant` directive as well. There is also no `PMC`, so the robustness directives are always turned off in production mode.

The functions get added to a specific object. So, there is no concept of definition generation or instantiation. A field is created with these two functions.

```
CreateField: function(where, name, def){},
RegisterField: function(name, def){},
```

R.6.1. CreateField

Description

The function creates the field in an object given as argument. If the object is null or undefined, the field is created in the window object.

Input Arguments

The function takes three arguments. The first argument is the object inside which the functions must be added. If this is null, it is replaced with the window object. The second argument is the name of the field. The third argument is the definition. This is a stripped down version of the class definition described in R.5.1.

Return Value

The function returns nothing.

The function throws an exception

- if there are variables defined
- if there are no functions defined
- if function use directives not intended for fields
- if the field object already has members with the same names as the functions

Remarks

The field is accessible only in local scope unless its exposed to outer scope via some mechanism.

R.6.2. RegisterField

Description

The function creates the field in an internal object and registers it.

Input Arguments

The function takes two arguments. The first argument is the name of the field. The second argument is the definition. This is a stripped down version of the class definition described in R.5.1.

Return Value

The function returns nothing.

The function throws an exception

- if there are variables defined
- if there are no functions defined
- if function use directives not intended for fields
- if the name is already registered

Remarks

The function adds the field object to a common database which makes it accessible in global scope. The function exists in two places, one in the environment object and one in a namespace object. The one in the environment adds the definition to the database directly while the one in the namespace disambiguates it before adding to the database. The registered definition can be retrieved via `GetObjectDef` function and its existence can be checked via `ObjectDefExist` function, both present the JEEP object

R.7. Group

A group is more versatile than a field and is nearly a class. It differs from a class in the following ways.

- it cannot have managed constructor, though it ca have constructor
- it cannot have destructor
- it cannot have static members
- the functions cannot use virtual and abstract directives
- it cannot extend from other groups

The functions and variables get added to a specific object. So, there is no concept of definition generation or instantiation. A group is created with these two functions.

```
CreateGroup: function(where, name, def){},
```

```
RegisterGroup: function(name, def){},
```

R.7.1. CreateGroup

Description

The function creates the group in the object given as argument. If the object is null or undefined, the group is created in the window object.

Input Arguments

The function takes three arguments. The first argument is the object inside which the functions must be added. If this is null, it is replaced with the window object. The second argument is the name of the group. The third argument is the definition. This is a stripped down version of the class definition described in R.5.1.

Return Value

The function returns nothing.

The function throws an exception

- if there are no functions defined
- if the definition uses properties not intended for groups
- if the group object already has members with the same names given in the definition

Remarks

The group is accessible only in local scope unless its exposed to outer scope via some mechanism.

R.6.2. RegisterGroup

Description

The function creates the group in an internal object and registers it.

Input Arguments

The function takes two arguments. The first argument is the name of the roup. The second argument is the definition. This is a stripped down version of the class definition described in R.5.1.

Return Value

The function returns nothing.

The function throws an exception

- if there are no functions defined
- if the definition uses properties not intended for groups

- if the name is already registered

Remarks

The function adds the group object to a common database which makes it accessible in global scope. The function exists in two places, one in the environment object and one in a namespace object. The one in the environment adds the definition to the database directly while the one in the namespace disambiguates it before adding to the database. The registered definition can be retrieved via `GetObjectDef` function and its existence can be checked via `ObjectDefExist` function, both present the JEEP object

R.7.3. Init

Description

Every group object gets this function added to the object if the group defines a constructor. This function is supposed to be called to invoke the constructor.

Input Arguments

The function takes any number of arguments.

Return Value

The function returns nothing.

Remarks

The function invokes the constructor with the same arguments as it gets. This function invokes the constructor only once no matter how many times it is called.

R.8. Namespace

A namespace is a mechanism to disambiguate names when registering objects. There is only one global database that stores all the registered objects and it is more than likely that names clash, especially when JEEP is used by multiple, disjoint library creating clients. A namespace provides a simple mechanism to solve the problem.

The `CreateNamespace` function of the environment returns an object that looks like this.

```
namespace = {
       GetEnvironment: function(){},
       RegisterRecordDef: function(name, def){},
       RegisterStructDef: function(name, def){},
       RegisterClassDef: function(name, def){},
       RegisterField: function(name, def){},
       RegisterGroup: function(name, def){},
       GetObjectDef: function(name){},
       ObjectDefExists: function(name){},
       Partition: function(names){},
       Flatten: function(){},
```

```
}
```

A namespace consists of a subset of the environment functions described in R.2 that deal with names. The end result of the registering functions is identical (internally the namespace functions just forward the calls to the environment functions), but the given names are transparently disambiguated suitably so as to not cause name clashes. This means the `GetObjectDef` and `ObjectDefExists` functions can directly use the names used in registering like the ones present in the JEEP object do and not care abut the disambiguating mechanism. The `GetEnvironment` function simply returns the environment object that create the namespace.

R.8.1. Partition

Description

This function partitions the namespace into sub namespaces which are accessible by name.

Input Arguments

This function takes a string which is a comma separated list of names. The number of names correspond to the number of partitions that will be made. Each name is associated with a part uniquely.

Return Value

This function returns nothing.

This functions throws exceptions

- if the names are repeated

Remarks

The names are only for accessing the specific sub namespaces and have no bearing on the name disambiguation mechanism that the namespace provides.

The object on which this function is invoked is itself portioned. Such a namespace will have a member named '$'. The partitions exist within this object and are accessible as normal properties of the object.

Each partition is itself a namespace, so partitioning can be done recursively to create a tree structure.

R.8.2. Flatten

Description

This function converts the tree like partitions structure in a namespace into an array for linear access.. On an namespace with no tree structure, this function has no noticeable effect.

Input Arguments

This function takes no arguments.

Result Value

This function returns an array containing all the parts. The flattening is done in a depth first post order traversal, where all descendents are listed before a node.

This function throws exception

- if the names of descendents clash

Remarks

This function doesn't affect the structure of the namespace itself, it just returns a new object, an array, containing the flat structure.

R.9. Library

A library is a higher level object that uses namespaces to logically group objects. Though conceptually there exists a library object, in practice its just a namespace object.

A library is an object that is always registered, and retrieved whenever necessary using the functions `RegisterLibrary` and `GetLibrary` respectively. Libraries exist in a different database, so a different retrieval function is necessary,

Apart from these two functions, there are two more functions dealing with library. However, these are not exposed anywhere else in the framework except inside the library constructor.

`R.9.1. RegisterLibrary`

Description

This function registers a library that can be retrieved later.

Input Arguments

This function takes two arguments. The first is a string for the name of the library. The second is a function that acts as the entry function to the library. This is intended to be the constructor analogue and build the library. The function can take any number of arguments and is completely user defined.

Return Value

This function returns nothing.

This function throws exception

- if the name is already registered

Remarks

This function exists in the main object instead of the environment because library registration has no behavioral changes imposed by environment.

R.9.2. RegisterLibraryBuilder

Description

This function registers a builder for a library.

Input Arguments

This function takes three arguments. The first is a string for the name of the builder, the second it the name of the library. The third is a function that acts as the entry function to the builder. The function can take any number of arguments and is completely user defined.

Return Value

This function returns nothing.

This function throws exception

- if the name is already registered
- if the library is not registered
- if a library exists with the builder name

R.9.3. GetLibrary

Description

This function is the `GetObjectDef` counterpart of the library and retrieves a registered library.

Input Arguments

This function takes one string argument for the name of the library.

Return Value

This function returns the library object, which is actually a namespace.

This function throws exception

- if the library was not registered

Remarks

When called for the first time, this function executes the constructor before returning the library object. Subsequent calls simple return the library object.

The constructor is setup with a `this` object that has four members – `env`, `namespace`, `Build` and `BuildPrivate`.

The `env` object is the same environment using which the `GetLibrary` was invoked, which caused the constructor to be executed. This is meant to be used to query the mode and client and setup object definitions accordingly.

The `namespace` is the implicit namespace that is supposed to be used to register the objects that make the library. This is what is returned by the function. For simple libraries, you can directly use the namespace functions, but for complex libraries with deep structures, it is better to automate it with the two functions `Build` and `BuildPrivate`.

R.9.3. Build

Description

This function is intended to build the library by calling the registered builders. Each builder can in turn further build via the same mechanism using the same function, which is available to that builder.

Input Arguments

This function takes a key value pair. The keys are strings, and signify the partitions on the namespace. The values are objects with have two properties – `builder` and `args`. The first is the name of a registered builder. The second is an array to be passed to the builder as arguments.

Return Value

The function returns nothing.

Remarks

The function first calls the `Partition` function on the implicit namespace, so refer to that function in R.8.1 for details on its behavior. Then, each builder is called with each partition setup as the builder's own implicit namespace.

If the key value is empty, it indicates that the current namespace itself must be used for building.

R.9.4. BuildPrivate

Description

This function is a generalization of the `Build` function. While the latter works on the implicit namespace, this function works on a specified namespace that is supposed to be private and local to the builder.

Input Arguments

This function one argument which is the same key value object as the `Build` function.

Return Value

This function returns a namespace.

Remarks

This function creates a namespace internally, builds using it and returns it.

This function is useful if a library needs to create local namespaces or use other libraries as an implementation detail.

R.10. Utilities

Unlike other aspects of Jeep, there is nothing about utilities to document any more concisely here than what is presented in he **Utilities** chapter. Any attempt to do so will result in a near copy of what is written there.

Appendix: Some Notes for C++ Developers

Only the most important parts are listed here. The numbering is only for convenience and has not other connotations.

A.1. Similarities

1. structure members are public by default
2. classes have constructors and destructors and are invoked in the C++ order
3. copy construction is available in the expected form
4. destructor is called when objects go out of scope
5. destructors are forbidden from throwing exceptions
6. classes can have public, protected and private members
7. classes can have static members with access restrictions
8. classes can have constant functions
9. classes can have one or more base classes
10. classes can have virtual and abstract (pure virtual) functions
11. classes with unimplemented abstract functions are not instantiated
12. virtual functions called from constructor and destructor behave as in C++

A.2. Differences

1. structures and classes cannot stand in for each other
2. structures can have only public and private members
3. structures cannot have static members
4. structures cannot extend other structures
5. there are no overloaded functions
6. class constructors can return string to indicate reason for failure of construction
7. class constructor and destructor are always public
8. upon failed construction, partial destruction is always attempted
9. duplicate member names are not allowed in class hierarchy
10. ambiguity resolution in hierarchy is only via wrapper classes
11. many inheritance patterns that are valid in C++ are invalid §7.3.1.
12. base classes are always virtual
13. base classes are merged into derived classes and lose their identity
14. virtual call from constructor and destructor can be treated as errors
15. a function defined as virtual should be defined as virtual throughout the hierarchy
16. a function defined as abstract should be defined as virtual when implemented, and henceforth as virtual throughout the hierarchy