Appendix: Coding Styles

C> ***The best laid schemes of mice and men Often go awry. (Roberts Burns)***

> ***"You can't call something reusable unless it's been reused at least three times on three separate projects by three separate teams. You can attempt to design for reuse, but you can't claim success until some element of the application has been reused at least three times. Reusability is in the eye of the beholder, not in the eye of the creator."*** from ***A Realistic Look at Object-Oriented Reuse***

Game Design System™ and the ***Game Recipes™*** are reusable. Game developers, for the past decade, have been migrating toward ***"Entities and Components"*** design. The deeply nested "OOP Class hierarchy" have proven too inflexible — no doubt you heard the ***"... banana, gorilla, jungle" story.*** Apple Gameplaykit has embraced the Entities and Components methods, and they have stated, ***"Inheritance-Based Architecture Hinders Game Design Evolution"***

A>### JS Objects: TL;DR
A>JavaScript has been plagued since the beginning with misunderstanding and awkwardness around its “prototypal inheritance” system, mostly due to the fact that “inheritance” isn’t how JS works at all, and trying to do that only leads to gotchas and confusions that we have to pave over with user-land helper libs. Instead, embracing that JS has “behavior delegation” (merely delegation links between objects) fits naturally with how JS syntax works, which creates more sensible code without the need of helpers.
A> …
A>When you set aside distractions like ***mixins***, ***polymorphism,*** composition, ***classes,*** constructors, and instances, and only focus on the objects that link to each other, you gain a powerful tool in behavior delegation that is easier to write, reason about, explain, and code-maintain. Simpler is better. JS is “objects-only” (OO). Leave the classes to those other languages!
A>
A>At this point of understanding, we should really ask ourselves: is the difficulty of ***expressing classes and inheritance in pure JavaScript*** a failure of the language (one which can ***temporarily*** be ***solved*** with ***user libraries*** and ultimately solved by ***additions to the language*** like class { .. } syntax), ***as many*** devs ***feel,*** or is it something deeper?
A>
A?Is it indicative of a more fundamental disparity, that we’re ***trying to do something in JS*** that it’s ***just not meant to do?***

Encoding Styles Of Mice & Men

In this new 4th edition, I am turning a corner and leaving JavaScript OOP (traditional class-ful design) behind. I've learned a lot as an author, and many have ***pointed out*** the ***"error of my ways"*** — "99% of all JS programmers are wrong" — and it seems I have led the masses to their death and destruction. I repent; I have decided to become a disciple of the radical path, and all the new code will reveal it. ***To summarize my new path, I quote another JavaScript disciple — Eric Elliot, "When I switched from C++ and Java to JavaScript and stopped using classes, it was like coming out of a dark tunnel into the light. ... If I only tell you that classical inheritance is bad, you probably won’t believe me. If I tell you stories about how it’s wreaked chaos in projects I’ve been involved in, it gives you the opportunity to feel the pain and gain a real understanding."***

{width:100%}

X>***Exercise:*** Read how to migrate your OOP game design ***into Pure Aggregation***

How does classical OOP work?

A>***Quoted from Wikipedia,*** "The claim is, that traditional ***object-oriented programming (OOP)*** design principles, result in poor data locality, more so if runtime polymorphism ***(dynamic dispatch)*** is used (which is ***especially problematic*** on ***some processors).*** Although OOP does superficially seem to ***organize code around data,*** the practice is quite different. OOP is actually about organizing ***source code*** around ***data types,*** rather than physically grouping individual fields and arrays in a format efficient for access by specific functions. It also often hides layout details ***under abstraction layers,*** while a data-oriented programmer wants to consider this first and foremost.

X>***Exercise:*** Read the ***criticism concerning OOP*** and alternate recommendations in ***Component-based***

X>***Exercise:*** Review the ***Comparison of various programming languages usage of "OO"***

X>***Exercise:*** Study ***A Realistic Look at Object-Oriented Reuse***

How does OOP work in JavaScript?

It ***DOESn't!*** JavaScript is a prototype-based language with delegation. If a property or method is not found locally, then JavaScripts move up the prototype chain until it **IS** found. This is not the same as OOP inheritance. It does ***NOT*** have "classes" — although ES6 attempts to make you believe it does — and does ***NOT*** behave as ordinary class-ful languages. JavaScript uses *"delegation"* to mimic "class-ful inheritance".

Senior software engineers are accustom to many different programming paradigms, and can easy switch various syntax. Attempting to force-fit JavaScript into a formal "Class-ful" OOAD language was either an attempt to quite the masses of inexperience programmers or appeal to the laziness of writing in a different paradigm.

X>***Exercise:*** Read ***Goodbye, Object Oriented Programming***

X>***Exercise:*** Review Mozilla Developers Network tutorials on ***JS and OO.***

JavaScript Coding Styles Examples:

Q>***Aren't classical inheritance and prototypical inheritance really the same thing, just a stylistic preference?***

Answer: NO! Read about the difference in ***Common Misconceptions About Inheritance in JavaScript***

References for further study:

- ***Essential JS Design Patterns***
- ***Elements of JavaScript Coding Style*** — contains some excellent guidelines that will help to improve your coding style (and hopefully understand the reasoning behind them).
- ***https://github.com/airbnb/javascript* —** Airbnb make theirs coding style-guide publicly available and it not only includes coding style recommendations, but also explains the justification behind them. It would certainly make a good starting point or template for your own style guide. You can also configure some code linters to help you stick to a particular style-guide.

Q>Quote from StackOver, "So far I saw three ways for creating an object in JavaScript, which way is best for creating an object and why?"

Answer: Found this article: ***3 ways to define a JavaScript Class***

That's a fair question since ***Phaser Gaming Framework*** prides itself on open standard integration and permitting the developer to create source code in their own style. JavaScript is a ***class-less language,*** however classes can be simulated using functions. ES2015 introduced native support for OOP into JavaScript, however, this feature is ***fondant icing*** (aka ***"syntactical sugar"***) over JavaScript's existing prototype-based inheritance model.

Experiment with the differences between Pure JavaScript and OOP-fondant at ***https://www.typescriptlang.org/play/***

A>***QUOTE from Phaser3 Devlog***: "Phaser 3 is now built entirely with webpack2. All of the code is being updated (or has been updated) to ***use CommonJS format modules.*** And webpack2 is managing the tree-shaking and package building of the whole thing. ***There are no grunt or gulp scripts to be seen anywhere, as we simply don't need them.*** On a side note I've also been using yarn for package management, and it's truly great! The speed is shockingly impressive."

You will discover that game authors use the ***Phaser JavaScript Game Framework*** in a variety of JavaScript styles and formatting. They chunk their game logic code into blocks (aka functions) to easy reuse the results across several game products. Each function block should adhere to ***"Single Responsibility Principle" (SRP)*** which means a function block does just one thing. Of course, moderation and common sense must be used when applying "SRP"; OOP often spirals ***into the ridiculous*** *and should be weighed against* **"SOLID" programming practice.** *You will know when your code has reached "la-la-land" by its* **"coupling"** *— the number one indicator of highly coupled functions is the ripple effect of* **bilateral dependencies,** *and when function blocks are* **constantly calling some other function(s)** *for specific data which in OOP is a violation of the "open-close" principle. Tightly coupled code is furthermore harder to maintain and reuse since it cannot be separated. You've heard the story about the* **"... banana, Gorilla, and jungle"** *and how to go about* **fixing it? Right? "... You could use an adapter pattern!"***

Singleton

{caption:"A class with only a single instance with global access points."}
```js

```
//v2.x.x
var game = new Phaser.Game(480, 320, Phaser.AUTO, null, game.Boot);
```
```

```
//v3.x.x
var game = new Phaser.Game(config);  OR
var game = Object.assign({},Phaser.Game(config));  OR
var game.prototype = Object.create(Phaser.prototype);
```

You can experiment with these in the browser console for clarification. ***Note:*** Some people write

A>Bar.prototype = Object.create(Foo.prototype);
A>as
A>Bar.prototype = new Foo();

Demonstration #1: Function Declarations

This a good style to use whenever you'd like to reuse the code content more than once. In contrast, nothing in an "anonymous function" can be reused; the variables could be, but only within the scope of the anonymous function. Since we are not going to "call" the Phaser.Game more than once, it is faster to use it as part of or inside an anonymous function.

{caption:"JS Function"}
```js

```
//Function declarations in JavaScript are hoisted to the top of the
//  enclosing function or global scope.
//You can use the function before you declared it.
//Use this method if you want to create several similar objects.
//In your example, Person (you should start the name with a capital
//letter) is called the constructor function.
//This is similar to classes in other Object Oriented (OO) languages.
function person(fname,career,age,eyecolor) {
    this.firstname=fname;
    this.career=career;
    this.age=age;
    this.eyecolor=eyecolor;
}

myRogueCharacter = new person("John","Ranger",25,"blue");
document.write(myRogueCharacter.firstname + " is " +
myRogueCharacter.age + " years old.");
```

```

## Function with Prototype Delegation

A fully initialized instance used for "copying or cloning" as understood by OOP programmers. ***Technically, nothing is copied in JavaScript; objects are referenced by delegation.***

```js

```
function Person(name){
    this.name = name;
}

//adds a new property after object was established.
Person.prototype.getName = function(){
    return this.name
}
```

```

Demonstration 2: Function as a Variable

Functions could be declared as variables. There really no difference between this and the above example. They both product the same outcomes and are even called in the same ways. It's just a "stylistic choice" and leans toward the function used as an object/method style.

{caption:"JS Function assigned to a variable"}
```js
var person = function person(fname,career,age,eyecolor) {
    this.firstname=fname;
    this.career=career;
    this.age=age;
    this.eyecolor=eyecolor;
}

myRogueCharacter = new person("John","Ranger",25,"blue");
document.write(myRogueCharacter.firstname + " is " +
myRogueCharacter.age + " years old.");
```

Demonstration 3: Object Literal

The most common approach is by defining a JavaScript function where we then create an object using the `new` keyword. `this` can be used to help define new properties and methods for the object as follows

This function also produces the outcome as seen in the examples above but with a much different path of execution. Whenever a function is used as a "method", the syntax and nesting is the most apparent because our Robot object and the `killAllEarthings` method is split. This gives us the opportunity to add more related actions methods into our Robot object, and provides ***SOLID "open/closure" aspect*** in our programming. To use the `killAllEarthings` method you have to enter this object and travel down to the function itself — **Robot.killAllEarthings.**

Using object literals works well with larger games with multiple categories. Objects are used to group similar features together to promote better code organization, and facilitate maintenance and future game feature upgrades.

{caption:"JS Object Literal"}
```js
// Use this method if you only need one object of a kind (such as a singleton).
// If you'd like this object inheriting from another,
//  then you have to use a constructor function though.
// Note that function expressions are not hoisted.

var Robot = {
    metal: "Titanium",
    killAllEarthlings: function() {
        alert("Exterminate Earthlings!");
    }
};

Robot.killAllEarthlings();
```

T>***Hint:*** "An object created by a `constructor` doesn't actually get a `.constructor` property to point to which function object (i.e., `constructor`) from which it was created. ***This is an extremely common misconception.*** Read more about this

Demonstration 4: Object Literal using Array syntax

```js
var gameObject = {};

gameObject['property Name1'] = value;
gameObject['property Name2'] = value;
//NOTE: how could you use this in Movement Tables
gameObject['method Name'] = function(){ /* function code here */ }
```

```

The advantage of using the array naming method is the option of dynamically named properties.

Along with the decision of which function syntax to use, you have the decision bout which development patterns to use.

- ***Functions and Closure patterns:*** "This pattern of utilizing closure allows you to encapsulate everything inside a single object and create a form of public API that can be used throughout the application while keeping all the other functions and variables local to the original object." ***Learning JavaScript, by Tim Wright page 206***
- ***Event-driven patterns:*** "This pattern is a little more straightforward because you set up functions and then call them with event listeners. It is especially good if you don't care about setting up a public API for your application. Event driven JavaScript uses a little less code when compared to other patterns, and many enjoy this method because it is thought to be short and to the point." ***Learning JavaScript, by Tim Wright page 207***

There are various ways to define a JavaScript object. It is totally based upon your game's requirements and how you intend to process information. This becomes an important issue when using Phaser v3.

## Deeper Dive: LOW LEVEL GARBAGE COLLECTION IN JS

***Research this article: "HOW TO WRITE LOW GARBAGE REAL-TIME JAVASCRIPT"***

## Phaser Coding Style debated {#codingStyle}

It was difficult learning how to create games when I first discovered Phaser JS Gaming Framework in 2013. The primary source of confusion was the various ways JavaScript allows complete freedom, and those that attempt to superimpose Object Oriented Program (OOP) on top of the JavaScript prototype language.

A>Quote from http://www.phpied.com/3-ways-to-define-a-javascript-class/: "JavaScript is a very flexible object-oriented language when it comes to syntax. ... It's important to note that ***there are no classes in JavaScript.*** Functions can be used to somewhat simulate classes, but in general ***JavaScript is a class-less language. Everything is an object. And when it comes to inheritance, objects inherit from objects, not classes from classes as in the "class"-ical languages."***

X>***Exercise:*** Read about JavaScript Anti-patterns in Learning JavaScript Design Patterns

X>***Exercise:*** Study the source code at and compare to the ***WARNING provided in Learning JavaScript Design Patterns concerning Anti-patterns.*** An excerpt follows:

A>
A>## Phaser Forum Discussion on Coding JS style:
A>http://www.html5gamedevs.com/topic/22865-to-prototype-or-not-to-prototype/
A>
A>**You don't have to code in an object-oriented way, and you don't have to use prototypes or `this` at all in JS.** JS supports many different ways of writing code and it tries to remain neutral about the 'best' way of doing things, largely because the 'best' way is not language dependent, ***it's based on the interpreter and on developer preference.***
A>
A>**Personally I'm not a fan of classical-style inheritance in JS, or the new and this keyword, and I'm not alone, check out the Gorilla/Banana problem.** Functional programming is a buzz word in JS right now, but it's not some new JS thing, JS has always been built around two fundamental coding paradigms:
A>
A> - **Prototypes**
A> - **Functions**
A>
A>And they don't exist in isolation from each other either!
A>
A>`this` is frequently misleading in JS and even at best it requires you to have knowledge of how the rest of the system works to be used (in most cases), **this cognitive load is bad.** We generally want to distill complex applications down into smaller and smaller chunks, to make them more manageable, which you can not do if you're carrying around the rest of the system in your brain (again, I could refer to that banana, sans gorilla and jungle please).
A>
A>Functional programming is a buzz word right now because it tries to address the inherent difficulties of dealing with scope and trying to set up
```

inheritance chains in a loosely typed language. Part of the problem is that everybody's first book on programming involves using functions, but then they pick up the 2nd and 3rd which deal with classical inheritance and they forget that first book because the 2nd and 3rd were so hard to learn and when you've invested a great deal into learning something its a normal human trait to be reticent to ditch it even when it's burning you.
A>
A>Just realized that this sounds perhaps unnecessarily aggressive towards using prototypes **(you can't actually avoid them in JS),** you need to know many different styles if you want to be a good programmer, just don't get hung up having to write classes or setting up prototype chains in JS, there are other methods, but try to understand the pros and cons of each approach, just saying ***'I dont like the way this or that looks'*** is **not** a valid argument, neither is ***'I prefer this method solely because it is more succinct'.***
A> ***. . . .***
A>another thing that could affect your coding style is the kind of editor you are using.
A>
A>If you are using an editor that supports auto-completion (like VS, Webstorm, SublimeText -with the Phaser plugin-, etc...) then you should take care of your style and provide "static" information via jsdoc and inheritance.
A>For example, this style:
A>
A> State*Boot.prototype = Object.create(State*Base.prototype);
A> State*Boot.prototype.preload = function () { ... };*
A>
A>is "better" than this other style:
A>
*A> State*Boot.prototype = {
A> preload: function () { ... }
A> };
A>
A>Why? Because in the first style the inference engine "knows" the class is extending a `Phaser.State` so you can type "this." and a list of methods from the base and current class appears.
A>
A>But in the second style, the inference engine has no way to know that the prototype is also a `Phaser.State`, so if you type "this." it shows only a list of Object methods.

Even at the dawn of Phaser v3.x.x, Rich Davey is drawing a "line in the sand" concerning the use of `ES5` or `ES6`.

A>### Class Wars — a quote from Newsletter 112
A>
A>We've recently had a few issues opened asking why we're using our own custom `Class` and not `ES6 classes`. To be honest the more recent issues have been getting a little abusive and it's starting to grate. So I'll explain once and for all and be done with it.
A>
A>Phaser 3 is written in `ES5`. All 56,000+ lines of it. We have every intention of migrating to ES6 later this year, but not now and certainly not this close to launch. In order to help both devs and ourselves, we use our own implementation of Class. It doesn't do much more than the old prototype method of creating classes that we used in v2, but it's tidier and gives us more structure internally. It's basically a bit of syntactic sugar that makes our lives easier. It's also exposed in the API so those of you who code in `ES5` and want to use it can do so if you wish.
A>
A>In order to use `ES6` classes instead, we would have to introduce `babel` into our workflow and build process. This is not something I'm willing to do when we're literally a few weeks out from release. The impact would not be insignificant for what is ultimately zero actual benefit.
A>
A>There appears to be a misunderstanding that because we use our own `Class` format that means you can't then use `ES6 classes` in your code. This is completely untrue. If you want to use `ES6` to make your games, go ahead, it works perfectly. You can use `native classes` that extend any Phaser objects without any problems at all. We know because we tested it months ago and it's still true today (because I re-tested it last week.)
A>
A>When we swap to `ES6` we will do it properly. I'm very much looking forward to using native classes, `destructured` objects, and default argument options. But there is a time and a place for doing this, and it absolutely isn't now.
A>

References:

- The Principles of Object-Oriented JavaScript by Nicholas C. Zakas
- ***Different ways to create objects and the resulting prototype chain***
- ***Object prototypes***

- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Inheritance
- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects
- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics